User's Guide and Documentation of the Parallel HO-PD Benchmark on the IBM SP2*

Masahiro Arakawa, Zhiwei Xu, and Kai Hwang
Parallel Computing Research Laboratory
University of Southern California
Los Angeles, CA 90089

June 19, 1995
CENG Technical Report 95-10

* This work was supported by a research subcontract from MIT Lincoln Laboratories to the University of Southern California as part of the ARPA Mountaintop Program. All rights are reserved by MIT/LL and by USC. Distribution is limited upon approval from the Mountaintop program at MIT/LL.

Abstract

The Higher-Order Post-Doppler (HO-PD) benchmark is one of the three programs in the parallel Space-Time Adaptive Processing (STAP) benchmark suite, jointly developed by the University of Southern California and MIT Lincoln Laboratory. This report provides an overview of the parallelization of the HO-PD benchmark for the IBM SP2 massively parallel processor. A user's guide, which lists the files and procedures making up the parallel HO-PD program as well as directions regarding the compilation and execution of the parallel HO-PD benchmark, is provided.

This project was conducted between August 1994 and May 1995 by the benchmark parallelization team at the University of Southern California: Kai Hwang, Zhiwei Xu, and Masahiro Arakawa. Hwang served as the principal investigator for the entire project. Xu developed the parallelization strategies for all the STAP benchmarks. Xu and Arakawa jointly developed the parallel code and collected performance measurements. Arakawa documented the parallel benchmark programs as reported here. This project was funded by the ARPA Mountaintop Program as a subcontract from MIT Lincoln Laboratory.

The STAP benchmarks allow us to evaluate the performance of a particular parallel computer for real-time digital radar signal processing by simulating the computations necessary for STAP. Digital adaptive signal processing is the one technology which has the potential to achieve thermal noise limited performance in the presence of jamming (electronic counter-measures) and clutter (signal returns from land) [Titi94]. A good description of the original HO-PD C source code can be found in [LL94].

We used a *single-program, multiple data stream* (SPMD) approach to parallelizing the benchmarks, directing each task to perform the same algorithm on independent portions of the data set. We added message-passing operations, such as broadcast and total exchange, to redistribute data. Because of the high message-passing overhead associated with the SP2, we adopted a coarse-grain strategy, minimizing the number of internode communication operations in our parallel programs.

Section 1 describes the computations performed in the HO-PD benchmark and its implementation in the sequential version of the program. Section 2 describes the parallel HO-PD benchmark on the SP2. Sections 3 through 6 contain the user's guide to the parallel program, listing the files and procedures making up the HO-PD program, as well as directions regarding the compilation and execution of the parallel benchmark.

An overview of the Mountaintop Program can be found in [Titi94]. Descriptions of SP2 message-passing operations can be found in [IBM94b]. More information about the message-passing performance of the SP2 can be found in [Xu95a]. Details of parallel application code development can be found in [Xu95b]. Additional algorithm analysis and performance results from this project can be found in [Arak95a, Hwan95a, Hwan95b, Hwan95c]. The paper [Hwan95c] presents a comprehensive report on the STAP benchmark performance results on the SP2. User's guides for the parallel APT (Adaptive Processing Testbed) and General benchmarks can be found in [Arak95b, Arak95c].

Contents

1 Sequential HO-PD Program

We generated the parallel HO-PD program by modifying the original sequential version. Below, Figure 1.1 shows the data flow of the HO-PD benchmark.

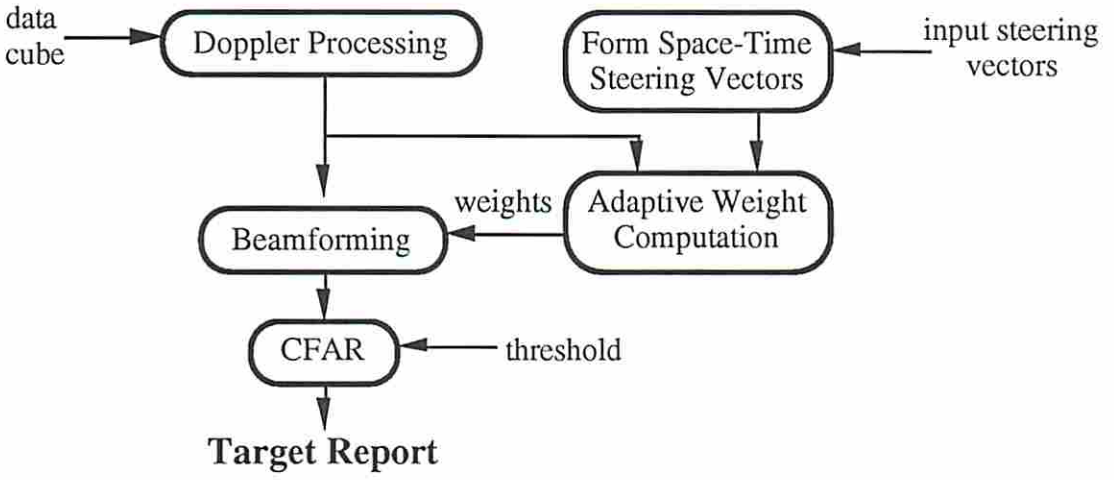


Figure 1.1: HO-PD benchmark data flow diagram

While analyzing the original sequential program, we developed a higher-level program skeleton to help us visualize the structure of the benchmark. Program skeletons are much shorter and simpler to understand than the complete code, while still displaying all the control and data dependence information needed for parallelization. The key words **`in`**, **`out`**, and **`inout`** are similar to those used in the Ada programming language, and indicate that a subroutine parameter is read only, write only, or read/write, respectively. The array index notation `[ . ]` is used to indicate that all array elements along that dimension are to be used. The program skeletons were adapted from [Hwan95c]. Below, Figure 1.2 shows the program skeleton for the sequential HO-PD program.

```
COMPLEX data_cube[PRI][RNG][EL], detect_cube[PRI][BEAM][RNG],
        twiddle[PRI];

/* Doppler Processing (DP) */
compute_twiddle_factor (out twiddle, in PRI);
for (j = 0; j < RNG; j++)
  for (k = 0; k < EL; k++)
    fft (inout data_cube[.][j][k], in twiddle, in PRI);

/* Beamforming (BF) */
for (i = 0; i < PRI; i++)
  beam_form (in data_cube[i][.][.], in data_cube[i-1 % PRI][.][.],
             in data_cube[i+1 % PRI][.][.], out detect_cube[i][.][.]);

/* Target Detection (CFAR) */
for (i = 0; i < PRI; i++)
  for (j = 0; j < BEAM; j++)
    compute_target (inout detect_cube[i][j][.]);

/* Target Report */
m = 0;
for (k = 0; k < RNG; k++)
  for (i = 0; i < PRI; i++)
    for (j = 0; j < BEAM; j++)
      if (IsTarget (in detect_cube[i][j][k]))
        {
          target_report[m].pri = i;
          target_report[m].beam = j;
          target_report[m].rng = k;
          target_report[m].power = detect_cube[i][j][k].real;
          m = m + 1;
          if (m > 25) goto finished;
        }
finished:
```

Figure 1.2: Sequential HO-PD program skeleton

Not included in this program skeleton, but implicit to the program, is the data file access step. In this step, the program reads the input data cube and the steering vectors from disk. The input data cube is stored in a packed binary format; prior to use by the program, it must be converted into floating-point format (see Figure 1.3). The steering vectors consist of both primary and auxiliary steering vectors, and are stored on disk in ASCII format.

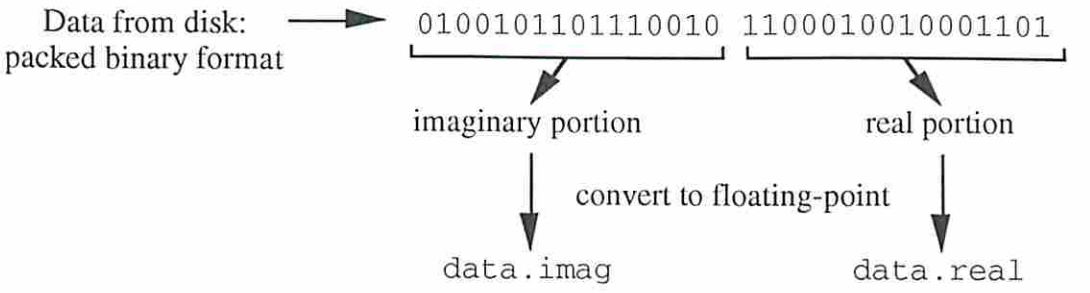


Figure 1.3: Packed binary format to floating-point format conversion

The program first transforms the input data cube into the Doppler domain by performing FFTs on the data. In this Doppler processing step, an FFT is performed along the PRI dimension on each column of constant EL and RNG (see Figure 1.4). The spatial steering vectors need to be similarly transformed to produce space-time steering vectors.

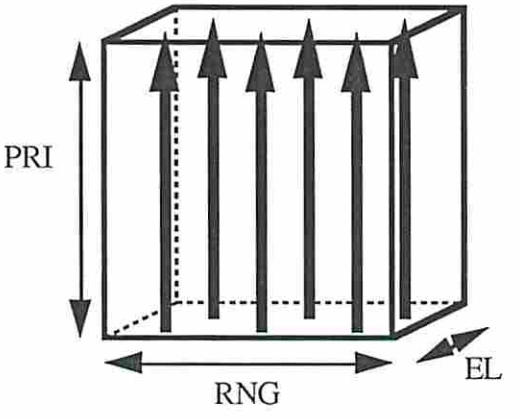


Figure 1.4: Dimension along which the FFTs are performed in the HO-PD benchmark

After the FFTs, the HO-PD performs the beamforming step. Each Doppler of interest is processed with its two adjacent (the preceding and succeeding) Dopplers to make $3 \times EL = DOF$ number of rows and RNG_S range gates. Every other range gate (i.e. 0, 2, 4, 6, etc.) to RNG_S (nominally 288) number of range gates is used.

A Householder transformation is performed on all DOF number of rows to lower triangularize all rows. Since there are NUMSEG segments of range gates, a Cholesky factorization and a forward back solve using the space-time steering vector on a segment for later application of weights to the segment adjacent to it are performed. The program starts at range gate 0 for the Cholesky factorization of the first segment, and applies the weights to the second segment starting at 0+RNGSEG. For the second or higher segments, the program starts at range gate = NUMSEG $\times$RNGSEG for the Cholesky factorization, and applies the weights to range gates starting at (NUMSEG-1) $\times$RNGSEG segment (the previous segment).

Finally the program performs the cell averaging CFAR target detection step. In this step, the program calculates the power of each range cell in a segment, as well as the cell average power of the range cells at least three range gates away from the cell of interest. The program compares these two powers, and if the cell power is above the threshold, then the cell is considered as having a target and added to the target list. This step is started at the lowest range so closer targets will be reported first.

2 Parallel HO-PD Program Development

In parallelizing the HO-PD program, we adopted the method described in [Xu95b]. This method (see Figure 2.1), which entails an early MPP performance prediction scheme, significantly reduced the parallel software development cost

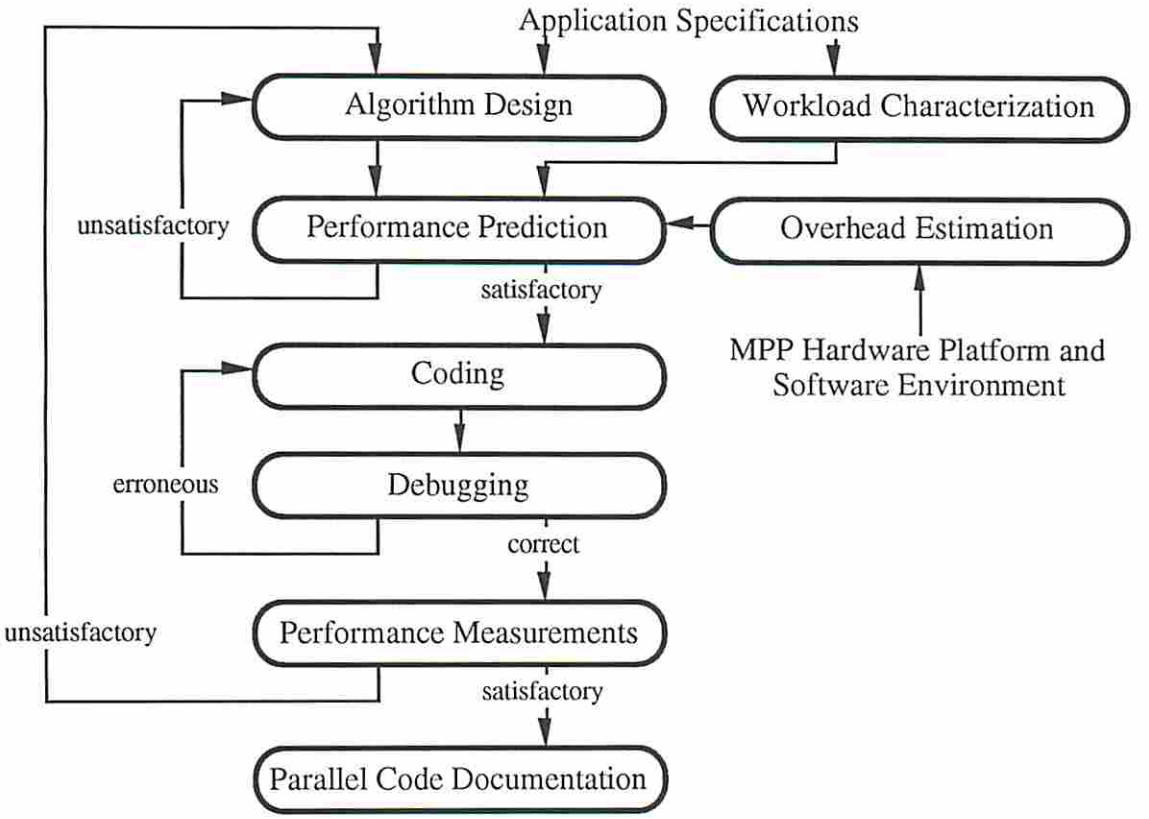


Figure 2.1: Early performance prediction based on workload and overhead characterization

During the algorithm design phase of the parallel program development, we considered several parallelization strategies [Hwan95c]. The nature of the algorithm in the HO-PD program made the simple compute-interact paradigm sufficient. In this paradigm, the parallel program is written as a sequence of alternating computation and

interaction steps. During a computation step, each task performs computations independently. During an interaction step, the tasks exchange data or synchronize. The parallel HO-PD program can be described as the following sequence of steps:

Compute:	Doppler Processing
Interact:	Total Exchange Circular Shift
Compute:	Beamforming Cell Averaging CFAR
Interact:	Target List Reduction

Because the same algorithm is applied to the entire data cube, we were able to use the SPMD paradigm. We exploited the parallelism inherent in the data set: in this parallel approach, each task performed the same algorithm on its portion of the input data set. The mapping of the parallel algorithm and the data set onto the SP2 is shown in Figure 2.2.

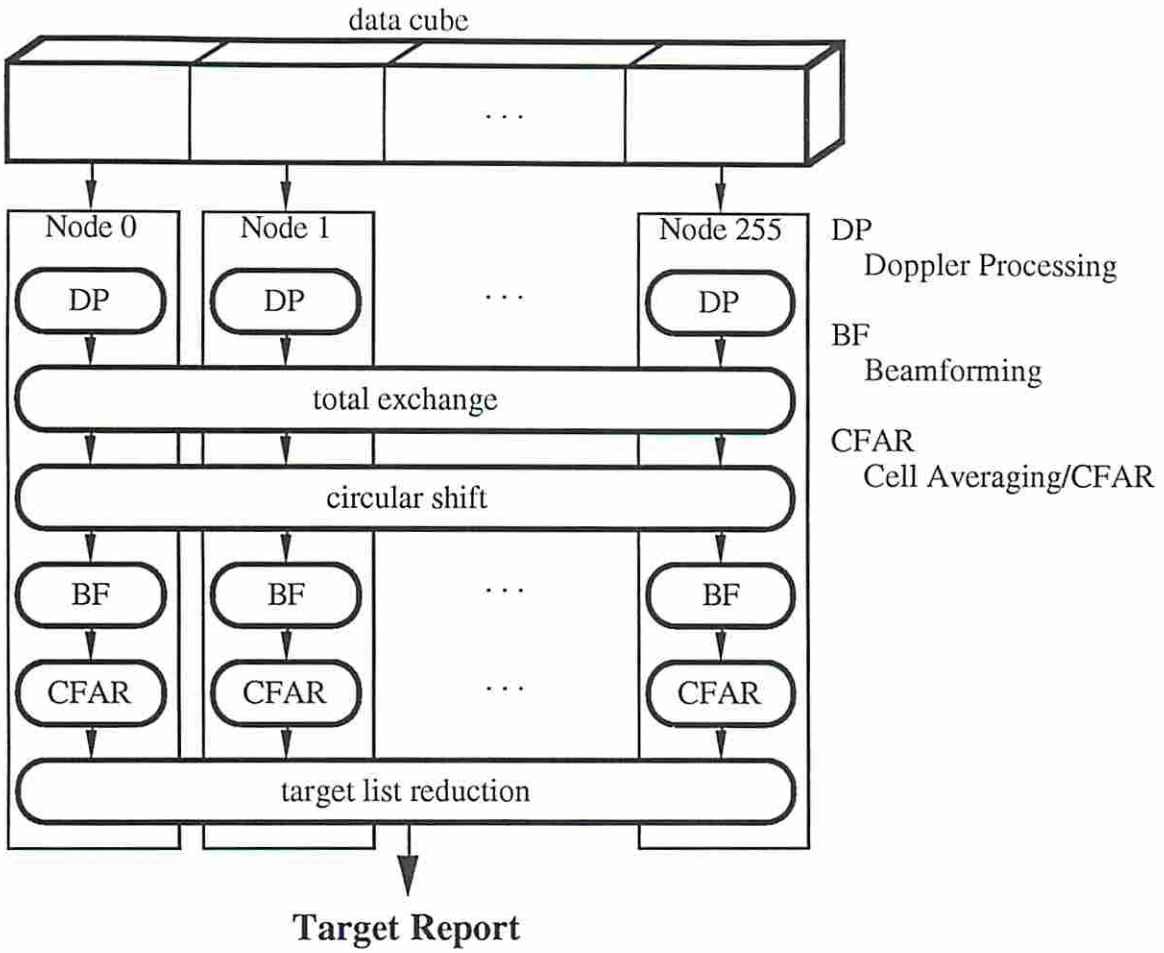


Figure 2.2: Mapping of the parallel algorithm and data set onto the SP2

During the coding step in the parallel software development process, we started with a program skeleton to help us visualize the structure of the benchmark. The parallel program skeleton is shown in Figure 2.3.

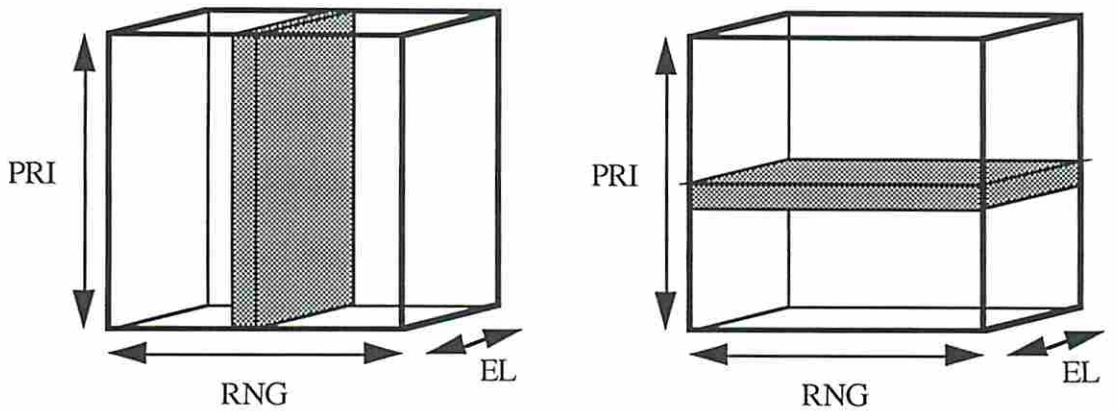
```
COMPLEX data_cube[PRI][RNG][EL], detect_cube[PRI][BEAM][RNG];

/* Doppler Processing (DP) */
parfor (j = 0; j < RNG; j++)  /* There are RNG tasks */
  {
    COMPLEX twiddle[PRI];  /* Local variable */
    compute_twiddle_factor (out twiddle, in PRI);
      /* Each task computes its own twiddle factor */
    for (k = 0; k < EL; k++)
      /* Each task computes EL FFTs */
      fft (inout data_cube[.][j][k], in twiddle, in PRI);
  }
barrier;
total_exchange data_cube[.][j][.] to data_cube[i][.][.];
shift data_cube[i][.][.] to data_cube[i-1 % PRI][.][.];
shift data_cube[i][.][.] to data_cube[i+1 % PRI][.][.];
barrier;
parfor (i = 0; i < PRI; i++)
  {  /* Beamforming (BF) */
    beam_form (in data_cube[i][.][.], in data_cube[i-1 % PRI][.][.],
               in data_cube[i+1 % PRI][.][.], out detect_cube[i][.][.]);
    barrier;
    /* Target Detection (CFAR) */
    for (j = 0; j < BEAM; j++)
      compute_target (inout detect_cube[i][j][.]);
    /* Target Report */
    m = 0;
    for (k = 0; k < RNG; k++)
      for (j = 0; j < BEAM; j++)
        if (IsTarget (in detect_cube[i][j][k]))
          {
            local_report[m].pri = i;
            local_report[m].beam = j;
            local_report[m].rng = k;
            local_report[m].power = detect_cube[i][j][k].real;
            m = m + 1;
            if (m > 25) goto finished;
          }
    finished:
  }
reduce local_reports to target_report;
```

Figure 2.3: Parallel HO-PD program skeleton

At the beginning of the parallel program, each task reads its portion of the data cube from disk in parallel. Because each FFT performed in the Doppler Processing step requires all the data along the PRI dimension (see Figure 1.4), but is independent of the

other FFTs along the RNG (as well as the EL) dimension, we divide the data cube equally along the RNG dimension (see Figure 2.4a).



(a) before total exchange (b) after total exchange

Figure 2.4: One task's slice of the data cube

In order to improve disk access performance, we modified the `read_input_STAP` procedure (see Appendix) so each task reads its entire slice of the data cube from disk before converting the data from a packed binary format (in which the data is stored on disk) to floating-point format (in which computations are done). In the sequential version, each complex number is read then converted immediately. Because the execution time of this step is not included in the total execution time of the parallel program (this step would not exist in the final implementation as part of an airborne radar processor), the performance of this step is not critical: the timer is started after the data is loaded and converted.

Each task performs FFTs on its slice of the data cube in parallel. In order to improve the performance of the FFTs, we recoded the FFT routine. Our version, a hybrid between the one in the original HO-PD program and a routine suggested by MHPCC, is a more efficient implementation of the same in-place FFT algorithm with bit reversal. By

replacing two separate arrays of floating-point numbers accessed by pointers with a single array of `COMPLEX` (a user-defined data structure) numbers, we were able to improve the processing rate on this routine from 15 MFLOPS per node to as high as 24 MFLOPS per node.

The remaining computational steps of the HO-PD benchmark require all the data along the RNG dimension, but have no data dependencies along the PRI and EL dimensions. Therefore, before continuing with the computation, the HO-PD program executes a total exchange operation to redistribute the data cube so that it is resliced along the RNG dimension (see Figure 2.4b). Because the MPL command `mpc_index` (which implements the total exchange operation) did not work quite to our specification, we needed to adjust the output of this operation. The output of the `mpc_index` operation is a 1-D vector, not the 3-D data cube slice the program needs. Therefore, we added the rewind step, which rewinds the 1-D vector back into a 3-D data cube slice.

So that we can give each task an equal slice of the data cube after the total exchange operation, we changed the number of range gates in the data set from its nominal value of 1250 to 1024. With 1024 range gates, the data can be divided along this dimension evenly by any power of two from 1 to 256. This adjustment affects the power levels of the targets slightly, but has no impact on the correctness of the parallel program. We confirmed this correctness by comparing the output of the sequential program using a data set with 1024 range gates to the output of the parallel program using the same data set.

The beamforming algorithm in the HO-PD program requires the preceding and succeeding Dopplers with each Doppler. Because these preceding and succeeding

Dopplers are located in the data cube slices of other tasks, they must be sent via message passing. We used two circular shifts to move the appropriate data (see Figure 2.5).

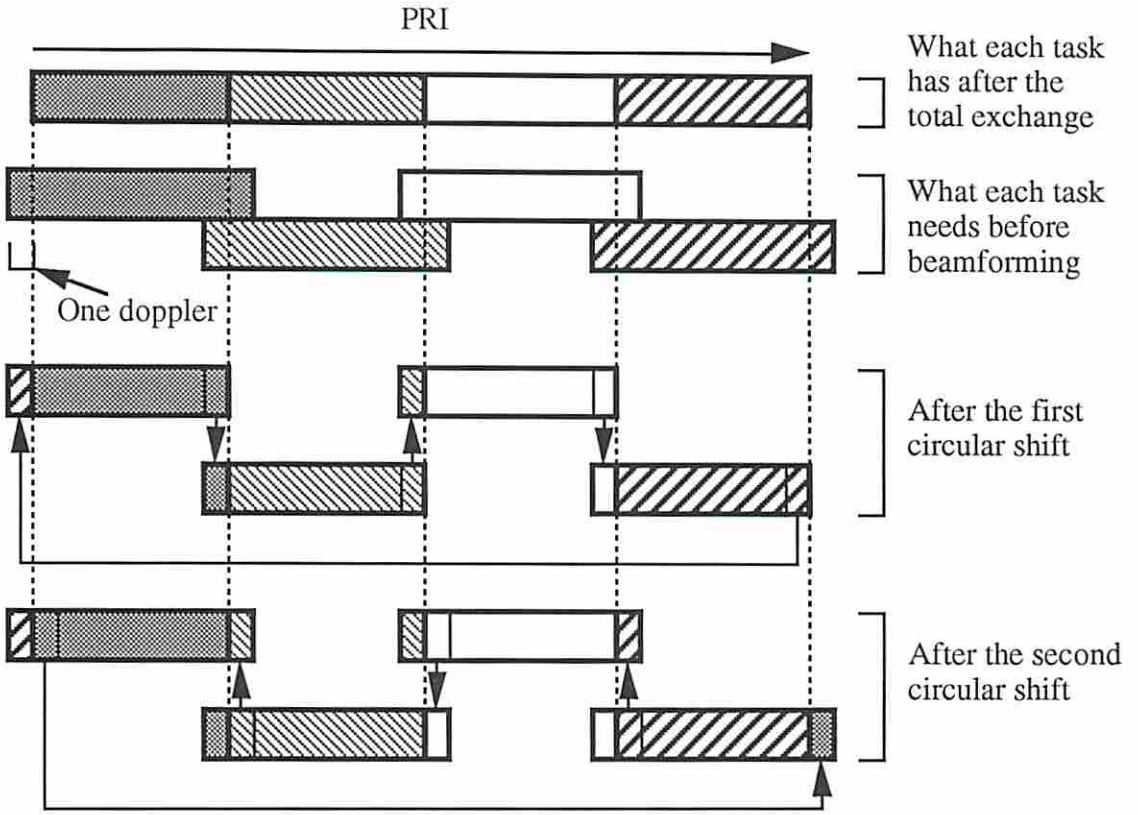


Figure 2.5: Circular shift operation in the parallel HO-PD program

Following the total exchange and circular shift operations, the HO-PD program performs the beamforming step. Because this algorithm displays data independence along the PRI and EL dimensions, each task can perform this step on its own portion of the data cube in parallel. The cell averaging CFAR step has been parallelized in a similar fashion. Each task generates its own target report, consisting of the closest targets found in its data cube slice, up to 25 targets.

In the target report reduction step, the parallel HO-PD program reduces the individual target reports into one final target report, consisting of the closest targets in the entire data cube, up to 25 targets. Several algorithms were studied, but only one was time-efficient [Hwan95c]. In this algorithm, the tasks pair off, then merge the two target lists, sorting by range and keeping up to 25 targets. The tasks holding the merged target lists then pair off, repeating this process until one task holds the target list for the entire data cube (see Figure 2.6).

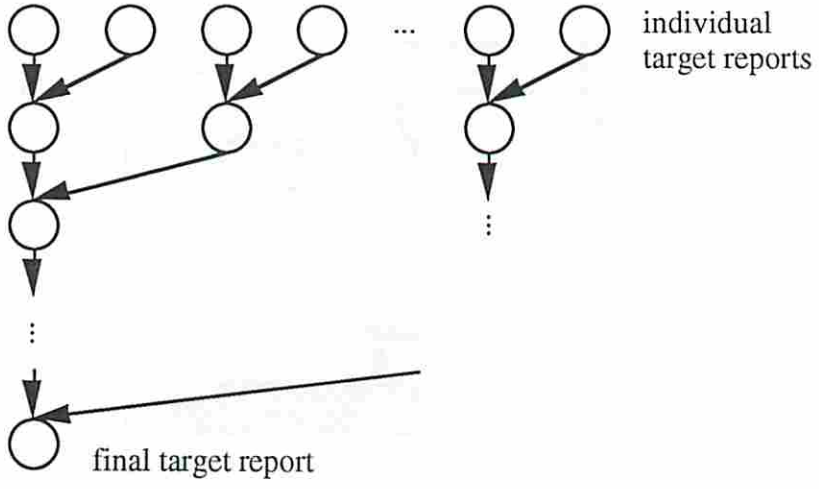


Figure 2.6: Target report merging process

Once the final target list is generated, the timer is stopped. The remainder of the program displays the final target list and collects execution time information. In order to improve the resolution and accuracy of the execution time measurements, we used two different timing calls: `times` and `gettimeofday`. The timing call `times` measures only CPU time used by the program, but has a resolution of only 10 ms. The timing call `gettimeofday` measures wall clock time (which may include time not spent on the HO-PD program), but has a resolution of 1 μs.

3 The Parallel HO-PD Code

The parallel program has been divided into the following files:

bench_mark_STAP.c	main ()
cell_avg_cfar.c	cell_avg_cfar ()
cmd_line.c	cmd_line ()
compute_beams.c	compute_beams ()
compute_weights.c	compute_weights ()
fft.c	fft (), bit_reverse ()
fft_STAP.c	fft_STAP ()
form_beams.c	form_beams ()
form_str_vecs.c	form_str_vecs ()
forback.c	forback ()
house.c	house ()
read_input_STAP.c	read_input_STAP ()

The purpose of each procedure is briefly described below:

main ()	This procedure represents the body of the parallel HO-PD program.
cell_avg_cfar ()	This procedure performs the cell averaging CFAR target detection on the slice of the detect cube local to a node.
cmd_line ()	This procedure extracts the names of the data file and the steering vector file from the command line.
compute_beams ()	This procedure performs beamforming on the node's slice of the data cube on a per-doppler basis.
compute_weights ()	This procedure computes the weights prior to beamforming. This procedure also generates a set of weights to be used in the beam computation routine from the modified space-time steering vectors.
fft ()	This procedure performs a single n-point in-place decimation-in-time FFT using n/2 complex twiddle factors. The implementation used in this procedure is a hybrid between the implementation used in the original sequential version of this program and the implementation suggested by MHPCC. This modification was made to improve the performance of the FFT on the SP2.

`bit_reverse ()`	This procedure performs a simple but somewhat inefficient bit reversal. This procedure is used by `fft ()`.
`fft_STAP ()`	This procedure performs an FFT along the PRI dimension for each column of data by calling `fft ()` (there are RNG $\times$ EL such columns).
`forback ()`	This procedure performs a forward and back substitution on an input array using the steering vectors, and normalizes the returned solution vector.
`form_beams ()`	This procedure performs the beamforming step by calling `compute_weights ()` and `compute_beams ()` on each doppler.
`form_str_vecs ()`	This procedure forms space-time steering vectors from the input spatial steering vectors.
`house ()`	This procedure performs an in-place Householder transform on a complex N $\times$ M input array, where M $\geq$ N. The results are returned in the same matrix as the input data.
`read_input_STAP ()`	This procedure reads the node's slice of the data cube from disk, then converts the data from the packed binary integer form in which it is stored on disk into floating-point format.

Figure 3.1 shows the calling relationship between the procedures.

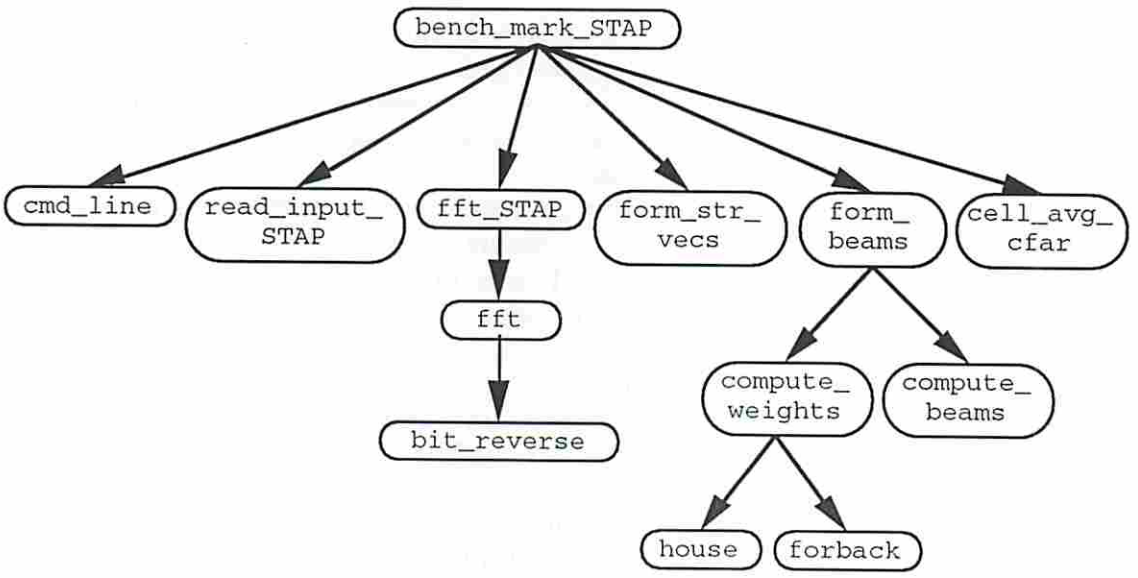


Figure 3.1: Calling relationship between the procedures in the parallel HO-PD program

The complete code listings for the IBM SP2 can be found in the Appendix.

4 The HO-PD Data Files

The input data cube file `new_stap.dat` is a modified version of the original data cube file `stap_data.dat`. We reordered the data in `new_stap.dat` so that each task can read its slice of the data cube from one contiguous location in the file. The modified data cube file still has PRI×RNG×EL complex numbers stored in 32-bit binary integer packed format. In this nominal data set, PRI = 128, RNG = 1024, and EL = 48. On disk, this data file is 48 MB.

The steering vectors file `new_stap.str` is identical to the original steering vectors file `stap_data.str`. The steering vectors file contains the number of PRIs in the data cube, the power threshold used in target detection, and the complex values for the steering vectors.

5 Compiling the Parallel HO-PD Program

The parallel HO-PD benchmark is compiled by calling the script file `compile_hopd`. This script file executes the following shell command:

```
mpcc -qarch=pwr2 -O3 -DSTAP -DIBM -o stap bench_mark_STAP.c
  cell_avg_cfar.c fft.c fft_STAP.c forback.c cmd_line.c house.c
  read_input_STAP.c form_beams.c compute_weights.c compute_beams.c
  form_str_vecs.c -lm
```

The `-qarch=pwr2` option directs the compiler to generate POWER2-efficient code. The `-O3` option selects the highest level of compiler optimization available on the `mpcc`

parallel C code compiler. The define flag `-DSTAP` must be included so that the compiler selects the proper data cube dimensions in the `defs.h` header file. The define flag `-DIBM` must be included so that the compiler can set the number of clock ticks per second (100 ticks per second on IBM machines, 60 ticks per second on SUN machines). To compile the program to use double-precision floating-point numbers instead of the default single-precision, add the define flag `-DDBLE`.

Prior to compiling the program, the user must set the number of nodes on which the program will be run. This number is defined as `NN` in the header file `defs.h`. This step is necessary because it is not possible to statically allocate arrays with their sizes defined by a variable in C. We have provided a set of header files with `NN` set to powers of two from 1 to 256. These header files are called `defs.001` through `defs.256`.

6 Running the Parallel HO-PD Program

The parallel HO-PD benchmark is invoked by calling the script file `run.###`, where `###` is the number of nodes on which the program will be run (and is equal to `NN` in `defs.h`). This script file executes the following shell command:

```
poe stap data_filename -procs ### -us
```

The command `poe` distributes the parallel executable file to all nodes and invokes the parallel program. The command line argument `data_filename` is the name of the files containing the input data cube and the steering vectors. The input data cube file must have the extension `.dat`, while the steering vectors file must have the extension `.str`. The command line arguments `-procs` and `-us` are flags to `poe`. The flag `-procs` sets the number of nodes on which the program will be run. The flag `-us` directs `poe` to

use the User Space communication library. This library gives the best communication performance. We have provided a set of script files to run the HO-PD benchmark, with the number of nodes equal to powers of two from 1 to 256. These files are called `run.001` through `run.256`.

Once the parallel program has begun and the timer has been started, the program will generate a " `Running...` " message. Once the program has finished and the timer has been stopped, it will generate a "`... done.`" message. The output of the program has the following format, shown in Figures 6.1 and 6.2. Figure 6.1 shows the target list, consisting of the targets found in the data cube, up to 25 targets. This number of targets was set as a parameter to the HO-PD benchmark. Figure 6.2 shows the timing report from the parallel HO-PD benchmark.

```
0:ENTRY     RANGE    BEAM    DOPPLER      POWER
0:   00       099      00       033       1.156658
0:   01       699      01       057       1.440849
0:   02       000      00       000       0.000000
0:   03       000      00       000       0.000000
0:   04       000      00       000       0.000000
0:   05       000      00       000       0.000000
0:   06       000      00       000       0.000000
0:   07       000      00       000       0.000000
0:   08       000      00       000       0.000000
0:   09       000      00       000       0.000000
0:   10       000      00       000       0.000000
0:   11       000      00       000       0.000000
0:   12       000      00       000       0.000000
0:   13       000      00       000       0.000000
0:   14       000      00       000       0.000000
0:   15       000      00       000       0.000000
0:   16       000      00       000       0.000000
0:   17       000      00       000       0.000000
0:   18       000      00       000       0.000000
0:   19       000      00       000       0.000000
0:   20       000      00       000       0.000000
0:   21       000      00       000       0.000000
0:   22       000      00       000       0.000000
0:   23       000      00       000       0.000000
0:   24       000      00       000       0.000000
```

Figure 6.1: Target list generated by the parallel HO-PD program

In Figure 6.1, the ENTRY column counts the number of targets detected, starting with target #0. The RANGE column indicates the range gate in which the target was found, and is proportional to the distance to the target. The BEAM column indicates the direction to the target. The DOPPLER column provides information regarding the speed of the target. The POWER column represents the power of the signal returned by the target.

The targets listed in Figure 6.1 are those targets placed in the input data cube when the data cube was generated. We confirmed the correctness of our parallel HO-PD program by comparing this target list to the target list generated by the original HO-PD benchmark (see [LL94]).

```
0:*** Timing information - numtask = 128
0:
0:  all_user_max      = 2.77 s, all_sys_max      = 0.03 s
0:  disk_user_max     = 0.05 s, disk_sys_max     = 0.03 s
0:  fft_user_max      = 0.09 s, fft_sys_max      = 0.00 s
0:  index_user_max    = 0.07 s, index_sys_max    = 0.00 s
0:  rewind_user_max   = 0.01 s, rewind_sys_max   = 0.00 s
0:  shift_user_max    = 0.09 s, shift_sys_max    = 0.00 s
0:  str_user_max      = 0.01 s, str_sys_max      = 0.00 s
0:  beam_user_max     = 1.10 s, beam_sys_max     = 0.00 s
0:  cfar_user_max     = 0.01 s, cfar_sys_max     = 0.00 s
0:  report_user_max   = 0.01 s, report_sys_max   = 0.00 s
0:
0:Wall clock timing -
0:  all_clock_time     = 3.039417
0:  disk_clock_time    = 0.324861
0:  fft_clock_time     = 0.051098
0:  index_clock_time   = 0.068951
0:  rewind_clock_time  = 0.005573
0:  shift_clock_time   = 0.072578
0:  str_clock_time     = 0.003732
0:  beam_clock_time    = 0.932793
0:  cfar_clock_time    = 0.001401
0:  report_clock_time  = 0.009694
```

Figure 6.2: Timing report generated by the parallel HO-PD program

In Figure 6.2, the timing entries ending with _user_max represent the largest amount of user CPU time spent by a node while performing that step, while the timing

entries ending with `_sys_max` represent the largest amount of system CPU time spent by a node while performing that step. The `Wall clock timing` entries indicate the amount of wall clock time spent on a given step, regardless of whether the time was spent by the program or by other programs.

The steps we timed are listed below:

`all`	This component is the end-to-end execution time of the entire program. This time is not equal to the sum of the times of the other steps, because this `all` time includes time spent waiting for synchronization not included in the individual steps.
`disk`	This component is the time to read the node's portion of the data cube from disk.
`fft`	This component is the time spent performing FFTs on this node's portion of the data cube.
`index`	This component is the time spent performing the total exchange operation.
`rewind`	This component is the time spent rewinding the 1-D output of the total exchange operation into a 3-D data cube slice.
`shift`	This component is the time spent performing the two circular shift operations.
`str`	This component is the time spent forming the space-time steering vectors.
`beam`	This component is the time spent performing the beamforming step.
`cfar`	This component is the time spent performing the cell averaging CFAR step.
`report`	This component is the time spent combining the individual target reports from all nodes into one final target report.

The time spent in these steps (T_{fft}, T_{index}, etc.) are combined to determine the total execution time of the parallel HO-PD benchmark:

$$T_{execution} = T_{fft} + T_{index} + T_{rewind} + T_{shift} + T_{str} + T_{beam} + T_{cfar} + T_{report} \qquad (6.1)$$

The HO-PD program's workload consists of 12.85 billion floating-point operations. By dividing this workload by the total execution time, we can calculate the sustained floating-point processing rate:

$$Sustained\ Processing\ Rate = \frac{Floating\text{-}point\ workload}{T_{execution}} \tag{6.2}$$

We define the system efficiency as the ratio between the sustained processing rate and the peak processing rate:

$$System\ Efficiency = \frac{Sustained\ Processing\ Rate}{Peak\ Processing\ Rate} \tag{6.3}$$

The timing results vary with machine size and from run to run, but the target list should be identical between runs and independent of machine size. To write these reports to a file, use standard UNIX output redirection:

```
run.### > out_filename
```

Both the parallel C source code and the data files have been stored on tape. Specifically, we archived them using the following UNIX command:

```
tar cvf /dev/rst8 commented
```

where `commented` is the name of the directory in which all the files are stored. The files can be extracted from the tape using the following UNIX command:

```
tar xvf /dev/rst8
```

The device specification (`/dev/rst8`) is machine specific.

Bibliography

[Arak95a] M. Arakawa, "Parallel STAP Benchmarks and Their Performance on the IBM SP2", Master's thesis, School of Engineering at the University of Southern California, August 1995

[Arak95b] M. Arakawa, Z. Xu, K. Hwang, "User's Guide and Documentation of the Parallel APT Benchmark on the IBM SP2", *Technical Report*, University of Southern California, June 1995

[Arak95c] M. Arakawa, Z. Xu, K. Hwang, "User's Guide and Documentation of the Parallel General Benchmark on the IBM SP2", *Technical Report*, University of Southern California, June 1995

[Hwan95a] K. Hwang, Z. Xu, M. Arakawa, "STAP Benchmark Performance on the IBM SP2 Massively Parallel Processor", *Proceedings of the Adaptive Sensor Array Processing Workshop 1995*, MIT Lincoln Laboratory, March 15-17, 1995, p. 75-91

[Hwan95b] K. Hwang, Z. Xu, M. Arakawa, "STAP Benchmark Performance of the IBM SP2 for Real-Time Signal Processing", submitted to *ACM/IEEE Supercomputing Conference '95, San Diego*, April 1, 1995

[Hwan95c] K. Hwang, Z. Xu, M. Arakawa, "Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing", submitted to *IEEE Transactions on Parallel and Distributed Systems*, May 11, 1995

[IBM94a] IBM Corp., *AIX Parallel Environment: Programing Primer*, Release 2.0, Pub. No. SH26-7223, IBM Corp., June 1994

[IBM94b] IBM Corp., *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*, Release 2.0, Pub. No. SH26-7228-01, IBM Corp., June 1994

[LL94] MIT Lincoln Laboratory, *Commercial Programmable Processor Benchmarks*, MIT Lincoln Laboratory, February 28, 1994

[Stun94] C. B. Stunkel, D. G. Shea, B. Abali, M. Atkins, C. A. Bender, D. G. Grice, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, P. R. Varker, "The SP2 Communication Subsystem", Technical Report, IBM Thomas J. Watson Research Center & IBM Highly Parallel Supercomputing Systems Laboratory, August 22, 1994

[Titi94] G. W. Titi, "An Overview of the ARPA/NAVY Mountaintop Program", *IEEE Adaptive Antenna Systems Symposium*, November 7-8, 1994

[Xu95a] Z. Xu, K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2 Multicomputer", submitted to *IEEE Parallel & Distributed Technology*, January 10, 1995

[Xu95b] Z. Xu, K. Hwang, "Early Prediction of MPP Performance by Workload and Overhead Quantification: A Case Study of the IBM SP2 System", submitted to *Parallel Computing*, April 1995

Appendix Parallel HO-PD Code

The parallel code for the HO-PD benchmark, along with the script to compile and run the parallel HO-PD program, are given in this appendix. The version of the code below is for 1 to 128 nodes.

A.1 bench_mark_STAP.c

```
/*
 * bench_mark_STAP.c
 */

/*
 * Parallel HO-PD Benchmark Program for the IBM SP2
 * ------------------------------------------------
 *
 * This parallel HO-PD benchmark program was written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern
 * California (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as
 * part of the ARPA Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 *
 * This file contains the procedure main (), and represents the body of the
 * HO-PD benchmark.
 *
 * The program's overall structure is as follows:
 *   1. Load data in parallel.
 *   2. Perform FFTs in parallel.
 *   3. Redistribute the data using the total exchange operation.
 *   4. Complete data redistribution using circular shift operations.
 *   5. Form a set of space-time steering vectors.
 *   6. Perform beamforming in parallel.
 *   7. Perform CFAR/cell averaging in parallel.
 *   8. Reduce local target lists into one complete target list.
 *
 * This program can be compiled by calling the shell script compile_hopd.
 * Because of the nature of the program, before the program is compiled, the
 * header file defs.h must be adjusted so that NN is set to the number of
 * nodes on which the program will be run. We have provided a defs.h file for
 * each power of 2 node size from 1 to 256, called defs.001 to defs.256.
 *
 * This program can be run by calling the shell script run.###, where ### is
 * the number of nodes on which the program is being run.
 *
 * Unlike the original sequential version of this program, the parallel HOPD
 * program does not support command-line arguments specifying the number of
 * times the program body should be executed, nor whether or not timing
 * information should be displayed. The program body will be executed once,
 * and the timing information will be displayed at the end of execution.
 *
 * The input data file on disk is stored in a packed format: Each complex
 * number is stored as a 32-bit binary integer; the 16 LSBs are the real
 * half of the number, and the 16 MSBs are the imaginary half of the number.
```

```
 * These numbers are converted into floating point numbers usable by this
 * benchmark program as they are loaded from disk.
 *
 * The steering vectors file has the data stored in a different fashion. All
 * data in this file is stored in ASCII format. The first two numbers in this
 * file are the number of PRIs in the data set and the threshold level,
 * respectively. Then, the remaining data are the steering vectors, with
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>
#include <mpproto.h>
#include <sys/time.h>
#include <time.h>



/*
 * Global variables
 * output_time: a flag; 1 = output timing information, 0 = don't output timing
 *   information
 * output_report: a flag; 1 = write data to disk, 0 = don't write data to disk
 * pricnt: the number of points in the FFT
 * threshold: minimum power return before a signal is considered a target
 * t_matrix: the T matrix
 */

int output_time = FALSE;
int output_report = FALSE;
int pricnt = PRI;
float threshold;
COMPLEX t_matrix[BEAM][EL];



/*
 * main ()
 *   inputs: argc, argv
 *   outputs: none
 *
 * This procedure is the body of the program.
 */

main (argc, argv)
     int argc;
     char *argv[];

{  /* main */

/*
 * Original variables (variables which were in the sequential version of the
 * HO-PD program)
 *
 * mod_str_vecs: holds the modified steering vectors
 * target_report: the 25 entries for targets which have been detected
 * el: the maximum number of elements in each PRI
 * det_bms: the number of beams in the detection data
 * beams: the total number of beams
 * rng: the maximum number of range gates in each PRI
 * f_temp: buffer for binary floating point output data
 * i, j, k: loop counters
 * dopplers: the number of dopplers after the FFT
 * xreal: storage for real FFT data
 * ximag: storage for imaginary FFT data
 * sintable: storage for SINE table
```

```
 * costable: storage for COSINE table
 * str_vecs: spatial steering vectors with EL COMPLEX elements
 * weight_vec: storage for the weight solution vectors
 * str_name: holder for the name of the steering vectors file
 * data_name: holder for the name of the input data file
 */

  static COMPLEX mod_str_vecs[BEAM][DOP][DOF];
  static float target_report[25][4];
  int el = EL;
  int det_bms = BEAM;
  int beams = BEAM;
  int rng = RNG;
  float f_temp[1];
  int i, j, k;
  int dopplers;
  float xreal[PRI];
  float ximag[PRI];
  float sintable[PRI];
  float costable[PRI];
  COMPLEX str_vecs[BEAM][EL];
  COMPLEX weight_vec[DOF];
  char str_name[LINE_MAX];
  char data_name[LINE_MAX];

  FILE *fopen();
  FILE *f_dop;
  extern void cmd_line();
  extern void fft_STAP();
  extern void form_str_vecs();
  extern void form_beams();
  extern void cell_avg_cfar();

/*
 * Variables we added or modified to parallelize the code
 *
 * rc: return code from MPL commands
 * numtask: the total number of tasks running this program
 * taskid: the identifier number for this task
 * nbuf[4]: a buffer to hold data from the MPL call mpc_task_query
 * allgrp: an identifier representing all the tasks running this program
 * n: loop counter
 * offset: a variable used to count the number of elements rewound so far
 * msglen: message length; used in MPL commands
 * out_row: no longer used
 * p, r, e: loop counters
 * blklen: block length; used in MPL commands
 * source: source task number
 * dest: destination task number
 * type: message type; used in MPL commands
 * nbytes: number of bytes in message
 * targets: number of targets
 * fft_out: local portion of the data cube
 * fft_vector: vector used during the total exchange operation
 * local_cube: local portion of the data cube after the total exchange and
 *   the circular shifts
 * shift_out: outgoing message of the circular shift operation
 * shift_in: incoming message of the circular shift operation
 * detect_cube: detection data cube
 */

  int rc;
  int numtask;
  int taskid;
  int nbuf[4];
  int allgrp;
  int n;
  int offset;
  int msglen;
```

```
  int out_row;
  int p, r, e;
  int blklen;
  int source;
  int dest;
  int type;
  int nbytes;
  int targets;

  static COMPLEX fft_out[PRI][RNG / NN][EL];
  static COMPLEX fft_vector[PRI * RNG * EL / NN];
  static COMPLEX local_cube[(PRI / NN) + 2][RNG][EL];
  static COMPLEX shift_out[RNG][EL];
  static COMPLEX shift_in[RNG][EL];
  static COMPLEX detect_cube[BEAM][DOP / NN][RNG];

/*
 * Timing variables
 *
 * *_start and *_end: the start and end CPU times for...
 * *_user and *_sys: the user and system time breakdowns for the time spent
 *   for...
 * *_user_max and *_sys_max: the largest user and system times for...
 * *_clock_start and *_clock_end: the start and end wall clock times for...
 * *_clock_time: the net wall clock time spent for...
 *
 *   all: the entire program
 *   disk: reading the data from disk
 *   fft: the FFTs
 *   index: the MPL command mpc_index
 *   rewind: rewinding the 1-D output of the mpc_index operation into a 3-D
 *     data cube
 *   shift: the two circular shift operations
 *   str: forming the steering vectors
 *   beam: the beamforming step
 *   cfar: the start and end time of the CFAR/cell averaging step
 *   report: the target reporting step
 */

  struct tms all_start, all_end;
  struct tms disk_start, disk_end;
  struct tms fft_start, fft_end;
  struct tms index_start, index_end;
  struct tms rewind_start, rewind_end;
  struct tms shift_start, shift_end;
  struct tms str_start, str_end;
  struct tms beam_start, beam_end;
  struct tms cfar_start, cfar_end;
  struct tms report_start, report_end;

  float all_user, all_sys;
  float disk_user, disk_sys;
  float fft_user, fft_sys;
  float index_user, index_sys;
  float rewind_user, rewind_sys;
  float shift_user, shift_sys;
  float str_user, str_sys;
  float beam_user, beam_sys;
  float cfar_user, cfar_sys;
  float report_user, report_sys;

  float all_user_max, all_sys_max;
  float disk_user_max, disk_sys_max;
  float fft_user_max, fft_sys_max;
  float index_user_max, index_sys_max;
  float rewind_user_max, rewind_sys_max;
  float shift_user_max, shift_sys_max;
  float str_user_max, str_sys_max;
  float beam_user_max, beam_sys_max;
```

```
  float cfar_user_max, cfar_sys_max;
  float report_user_max, report_sys_max;

  struct timeval all_clock_start, all_clock_end;
  struct timeval disk_clock_start, disk_clock_end;
  struct timeval fft_clock_start, fft_clock_end;
  struct timeval index_clock_start, index_clock_end;
  struct timeval rewind_clock_start, rewind_clock_end;
  struct timeval shift_clock_start, shift_clock_end;
  struct timeval str_clock_start, str_clock_end;
  struct timeval beam_clock_start, beam_clock_end;
  struct timeval cfar_clock_start, cfar_clock_end;
  struct timeval report_clock_start, report_clock_end;

  float all_clock_time;
  float disk_clock_time;
  float fft_clock_time;
  float index_clock_time;
  float rewind_clock_time;
  float shift_clock_time;
  float str_clock_time;
  float beam_clock_time;
  float cfar_clock_time;
  float report_clock_time;

/*
 * Begin function body: main ()
 *
 * Initialize for parallel processing: Here, each task or node determins its
 * task number (taskid) and the total number of tasks or ndoes running this
 * program (numtask) by using the MPL call mpc_environ. Then, each task
 * determins the identifier for the group which encompasses all tasks or
 * nodes running this program. This identifier (allgrp) is used in collective
 * communication or aggregated computation operations, such as mpc_index.
 */

  rc = mpc_environ (&numtask, &taskid);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_environ.\n");
      exit (-1);
    }

  if (numtask != NN)
    {
      printf ("Error - task number mismatch... check defs.h.\n");
      exit (-1);
    }

  rc = mpc_task_query (nbuf, 4, 3);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_task_query.\n");
      exit (-1);
    }
  allgrp = nbuf[3];

  if (taskid == 0)
    {
      printf ("Running...\n");
    }

  gettimeofday (&all_clock_start, (struct timeval*) 0);
  times (&all_start);

/*
 * Get arguments from the command line. In the sequential version of the
 * program, the following procedure was used to extract the number of times
 * the main computational body (after the FFT) was to be repeated, and flags
```

```
 * regarding the amount of reporting to be done during and after the program
 * was run. In this parallel program, there are no command line arguments to
 * be extracted except for the name of the file containing the data cube.
 */

  cmd_line (argc, argv, str_name, data_name);

/*
 * Read input files. In this section, each task loads its portion of the
 * data cube from the data file.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  loading data...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&disk_clock_start, (struct timeval*) 0);
  times (&disk_start);
  read_input_STAP (data_name, str_name, str_vecs, fft_out);
  times (&disk_end);
  gettimeofday (&disk_clock_end, (struct timeval*) 0);

/*
 * FFT: In this section, each task performs FFTs along the PRI dimension
 * on its portion of the data cube. The FFT implementation used in this
 * program is a hybrid between the original implementation found in the
 * sequential version of this program and a suggestion given to us by MHPCC.
 * This change in implementation was done to improve the performance of the
 * FFT on the SP2.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  running parallel FFT...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&fft_clock_start, (struct timeval*) 0);
  times (&fft_start);
  fft_STAP (fft_out);
  times (&fft_end);
  gettimeofday (&fft_clock_end, (struct timeval*) 0);

/*
 * Perform index operation to redistribute the data cube. Before the index
 * operation, the data cube was sliced along the RNG dimension, so each
 * task got all the PRIs. After the index operation, the data cube is sliced
 * along the PRI dimension, so each task gets all the RNGs.
 *
 * Because the MPL command mpc_index doesn't work to our specifications,
 * we need to rewind the 1-D matrix which is the output of the mpc_index
 * operation back into a 3-D data cube slice.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  indexing data cube...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&index_clock_start, (struct timeval*) 0);
```

```
  times (&index_start);
  rc = mpc_index (fft_out, fft_vector,
                  PRI * RNG * EL * sizeof (COMPLEX) / (NN * NN), allgrp);
  times (&index_end);
  gettimeofday (&index_clock_end, (struct timeval*) 0);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_index.\n");
      exit (-1);
    }

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  rewinding data cube...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&rewind_clock_start, (struct timeval*) 0);
  times (&rewind_start);
  offset = 0;
  for (n = 0; n < NN; n++)
    {
      for (p = 0; p < PRI / NN; p++)
       {
         for (r = n * RNG / NN; r < (n + 1) * RNG / NN; r++)
           {
             for (e = 0; e < EL; e++)
              {
                local_cube[p + 1][r][e].real = fft_vector[offset].real;
                local_cube[p + 1][r][e].imag = fft_vector[offset].imag;
                offset++;
              }
           }
       }
    }
  times (&rewind_end);
  gettimeofday (&rewind_clock_end, (struct timeval*) 0);

/*
 * Beamforming requires not only the dopplers made available after indexing,
 * but also the first doppler on both sides. Use two circular shift operations
 * to get them.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  performing circular shift...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&shift_clock_start, (struct timeval*) 0);
  times (&shift_start);
  msglen = RNG * EL * sizeof (COMPLEX);
  for (r = 0; r < RNG; r++)
    {
      for (e = 0; e < EL; e++)
       {
         shift_out[r][e].real = local_cube[PRI / NN][r][e].real;
         shift_out[r][e].imag = local_cube[PRI / NN][r][e].imag;
       }
    }

  rc = mpc_shift (shift_out, shift_in, msglen, 1, 0, allgrp);
  if (rc == -1)
    {
```

```
      printf ("Error - unable to call mpc_shift.\n");
      exit (-1);
    }

  for (r = 0; r < RNG; r++)
    {
      for (e = 0; e < EL; e++)
       {
         local_cube[0][r][e].real = shift_in[r][e].real;
         local_cube[0][r][e].imag = shift_in[r][e].imag;
       }
    }

  for (r = 0; r < RNG; r++)
    {
      for (e = 0; e < EL; e++)
       {
         shift_out[r][e].real = local_cube[1][r][e].real;
         shift_out[r][e].imag = local_cube[1][r][e].imag;
       }
    }

  rc = mpc_shift (shift_out, shift_in, msglen, -1, 0, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_shift.\n");
      exit (-1);
    }

  for (r = 0; r < RNG; r++)
    {
      for (e = 0; e < EL; e++)
       {
         local_cube[(PRI / NN) + 1][r][e].real = shift_in[r][e].real;
         local_cube[(PRI / NN) + 1][r][e].imag = shift_in[r][e].imag;
       }
    }
  times (&shift_end);
  gettimeofday (&shift_clock_end, (struct timeval*) 0);

/*
 * Form a set of space_time steering vectors composed of the concatenation of
 * doppler_transformed spatial steering vectors for doppler of interest and
 * preceding and succeeding dopplers. Modified steering vectors will be put in
 * the mod_str_vecs matrix.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  forming steering vectors...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&str_clock_start, (struct timeval*) 0);
  times (&str_start);
  form_str_vecs (str_vecs, mod_str_vecs);
  times (&str_end);
  gettimeofday (&str_clock_end, (struct timeval*) 0);

/*
 * Perform adaptive beamforming. Each task performs this step on its slice
 * of the data cube in parallel.
 */

/*
 *  if (taskid == 0)
 *    {
```

```
*       printf ("  beamforming...\n");
*     }
*/

  dopplers = pricnt / NN;
  mpc_sync (allgrp);
  gettimeofday (&beam_clock_start, (struct timeval*) 0);
  times (&beam_start);
  form_beams (dopplers, local_cube, detect_cube, mod_str_vecs);
  times (&beam_end);
  gettimeofday (&beam_clock_end, (struct timeval*) 0);

/*
 * Perform cell_avg_cfar and target detection. Each task performs this step
 * on its slice of the data cube in parallel.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  performing cell_avg_cfar...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&cfar_clock_start, (struct timeval*) 0);
  times (&cfar_start);
  cell_avg_cfar (threshold, dopplers, det_bms, rng, target_report,
                 detect_cube);
  times (&cfar_end);
  gettimeofday (&cfar_clock_end, (struct timeval*) 0);

/*
 * Gather target reports: In this step, the tasks collect the closest 25
 * targets. This target sorting is performed in the following manner. The tasks
 * pair off. The two tasks in this pair combine their target lists (i.e. the
 * targets found in their own slice of the data cube). Then, one of these two
 * tasks sorts the list, and takes the closest targets (up to 25). Then, the
 * tasks with these new, combined target lists pair off, and this process is
 * repeated again. At the end, one task will have the final target list, which
 * contains the closest targets in the entire data cube (up to 25).
 *
 * The target list combining is done by matched blocking-sends and blocking-
 * receives.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  gathering target reports...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&report_clock_start, (struct timeval*) 0);
  times (&report_start);
  for (i = 1; i < NN; i = 2 * i)
    {  /* for (i...) */
      blklen = 25 * 4 * sizeof (float);
      source = taskid + i;
      dest = taskid - i;
      type = i;
      if ((taskid % (2 * i)) == 0 && (NN != 1))
       {
         rc = mpc_brecv (target_report + 25, blklen, &source, &type,
                         &nbytes);
         if (rc == -1)
           {
             printf ("Error - unable to call mpc_brecv.\n");
```

```
            exit(-1);
          }
      }

      if ((taskid % (2 * i)) == i && (NN != 1))
      {
        rc = mpc_bsend (target_report, blklen, dest, type);
        if (rc == -1)
          {
            exit(-1);
          }
      }

/*
 *  Sort combined target list.
 */
      for (targets = 0; targets < 50; targets++)
      {  /* for (targets...) */
        for (r = targets + 1 ; r < 50 ; r ++)
          {  /* for (r...) */
            if (((target_report[targets][0] > target_report[r][0])
                && (target_report[r][0] > 0.0))
               || ((target_report[targets][0] < target_report[r][0])
                   && (target_report[targets][0] == 0.0)))
             {  /* if (...) */
               float tmp;

               tmp = target_report[r][0];
               target_report[r][0] = target_report[targets][0];
               target_report[targets][0] = tmp;
               tmp = target_report[r][1];
               target_report[r][1] = target_report[targets][1];
               target_report[targets][1] = tmp;
               tmp = target_report[r][2];
               target_report[r][2] = target_report[targets][2];
               target_report[targets][2] = tmp;
               tmp = target_report[r][3];
               target_report[r][3] = target_report[targets][3];
               target_report[targets][3] = tmp;
             }  /* if (...) */
          }  /* for (r...) */
      }  /* for (targets...) */
    }  /* for (i...) */
  times (&report_end);
  times (&all_end);
  gettimeofday (&report_clock_end, (struct timeval*) 0);
  gettimeofday (&all_clock_end, (struct timeval*) 0);

  if (taskid == 0)
    {
      printf ("... done.\n");
    }

/*
 * Now, the program is done with the computation. All that remains to be done
 * is to report the targets that were found, and to report the amount of time
 * each step took. The target list should be identical from run to run, since
 * the program started with the same input data cube. This target list and
 * execution time data reporting is performed by task 0.
 */

  if (taskid == 0)
    {
      printf ("ENTRY    RANGE    BEAM    DOPPLER      POWER\n");
      for (i = 0; i < 25; i++)
       printf ("  %.02d       %.03d       %.02d       %.03d       %f\n", i,
             (int)target_report[i][0], (int)target_report[i][1],
             (int)target_report[i][2], target_report[i][3] );
    }
```

```
/*
 * Collect timing information. Calculate elapsed user and system CPU time for
 * each step.
 */

  all_user = (all_end.tms_utime - all_start.tms_utime) / CLK_TICK;
  all_sys = (all_end.tms_stime - all_start.tms_stime) / CLK_TICK;
  disk_user = (disk_end.tms_utime - disk_start.tms_utime) / CLK_TICK;
  disk_sys = (disk_end.tms_stime - disk_start.tms_stime) / CLK_TICK;
  fft_user = (fft_end.tms_utime - fft_start.tms_utime) / CLK_TICK;
  fft_sys = (fft_end.tms_stime - fft_start.tms_stime) / CLK_TICK;
  index_user = (index_end.tms_utime - index_start.tms_utime) / CLK_TICK;
  index_sys = (index_end.tms_stime - index_start.tms_stime) / CLK_TICK;
  rewind_user = (rewind_end.tms_utime - rewind_start.tms_utime) / CLK_TICK;
  rewind_sys = (rewind_end.tms_stime - rewind_start.tms_stime) / CLK_TICK;
  shift_user = (shift_end.tms_utime - shift_start.tms_utime) / CLK_TICK;
  shift_sys = (shift_end.tms_stime - shift_start.tms_stime) / CLK_TICK;
  str_user = (str_end.tms_utime - str_start.tms_utime) / CLK_TICK;
  str_sys = (str_end.tms_stime - str_start.tms_stime) / CLK_TICK;
  beam_user = (beam_end.tms_utime - beam_start.tms_utime) / CLK_TICK;
  beam_sys = (beam_end.tms_stime - beam_start.tms_stime) / CLK_TICK;
  cfar_user = (cfar_end.tms_utime - cfar_start.tms_utime) / CLK_TICK;
  cfar_sys = (cfar_end.tms_stime - cfar_start.tms_stime) / CLK_TICK;
  report_user = (report_end.tms_utime - report_start.tms_utime) / CLK_TICK;
  report_sys = (report_end.tms_stime - report_start.tms_stime) / CLK_TICK;

/*
 * Calculate elapsed wall clock time for each step.
 */

  all_clock_time = (float) (all_clock_end.tv_sec
                          - all_clock_start.tv_sec)
    + (float) ((all_clock_end.tv_usec
              - all_clock_start.tv_usec) / 1000000.0);

  disk_clock_time = (float) (disk_clock_end.tv_sec
                           - disk_clock_start.tv_sec)
    + (float) ((disk_clock_end.tv_usec
              - disk_clock_start.tv_usec) / 1000000.0);

  fft_clock_time = (float) (fft_clock_end.tv_sec
                          - fft_clock_start.tv_sec)
    + (float) ((fft_clock_end.tv_usec
              - fft_clock_start.tv_usec) / 1000000.0);

  index_clock_time = (float) (index_clock_end.tv_sec
                            - index_clock_start.tv_sec)
    + (float) ((index_clock_end.tv_usec
              - index_clock_start.tv_usec) / 1000000.0);

  rewind_clock_time = (float) (rewind_clock_end.tv_sec
                             - rewind_clock_start.tv_sec)
    + (float) ((rewind_clock_end.tv_usec
              - rewind_clock_start.tv_usec) / 1000000.0);

  shift_clock_time = (float) (shift_clock_end.tv_sec
                            - shift_clock_start.tv_sec)
    + (float) ((shift_clock_end.tv_usec
              - shift_clock_start.tv_usec) / 1000000.0);

  str_clock_time = (float) (str_clock_end.tv_sec
                          - str_clock_start.tv_sec)
    + (float) ((str_clock_end.tv_usec
              - str_clock_start.tv_usec) / 1000000.0);

  beam_clock_time = (float) (beam_clock_end.tv_sec
                           - beam_clock_start.tv_sec)
    + (float) ((beam_clock_end.tv_usec
```

```
            - beam_clock_start.tv_usec) / 1000000.0);

  cfar_clock_time = (float) (cfar_clock_end.tv_sec
                           - cfar_clock_start.tv_sec)
    + (float) ((cfar_clock_end.tv_usec
            - cfar_clock_start.tv_usec) / 1000000.0);

  report_clock_time = (float) (report_clock_end.tv_sec
                             - report_clock_start.tv_sec)
    + (float) ((report_clock_end.tv_usec
            - report_clock_start.tv_usec) / 1000000.0);

/*
 * Find the largest amount of user and system CPU time spent using the
 * mpc_reduce call.
 */

  rc = mpc_reduce (&all_user, &all_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&all_sys, &all_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&disk_user, &disk_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&disk_sys, &disk_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&fft_user, &fft_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&fft_sys, &fft_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&index_user, &index_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
```

```
rc = mpc_reduce (&index_sys, &index_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

rc = mpc_reduce (&rewind_user, &rewind_user_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&rewind_sys, &rewind_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

rc = mpc_reduce (&shift_user, &shift_user_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&shift_sys, &shift_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

rc = mpc_reduce (&str_user, &str_user_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&str_sys, &str_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

rc = mpc_reduce (&beam_user, &beam_user_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&beam_sys, &beam_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

rc = mpc_reduce (&cfar_user, &cfar_user_max, sizeof (float), 0, s_vmax,
```

```
                allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&cfar_sys, &cfar_sys_max, sizeof (float), 0, s_vmax,
                allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&report_user, &report_user_max, sizeof (float), 0, s_vmax,
                allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&report_sys, &report_sys_max, sizeof (float), 0, s_vmax,
                allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

/*
 * Display timing information.
 */

  if (taskid == 0)
    {
      printf ("\n\n\n*** Timing information - numtask = %d\n\n", NN);
      printf ("  all_user_max     = %.2f s, all_sys_max     = %.2f s\n",
            all_user_max, all_sys_max);
      printf ("  disk_user_max    = %.2f s, disk_sys_max    = %.2f s\n",
            disk_user_max, disk_sys_max);
      printf ("  fft_user_max     = %.2f s, fft_sys_max     = %.2f s\n",
            fft_user_max, fft_sys_max);
      printf ("  index_user_max   = %.2f s, index_sys_max   = %.2f s\n",
            index_user_max, index_sys_max);
      printf ("  rewind_user_max  = %.2f s, rewind_sys_max  = %.2f s\n",
            rewind_user_max, rewind_sys_max);
      printf ("  shift_user_max   = %.2f s, shift_sys_max   = %.2f s\n",
            shift_user_max, shift_sys_max);
      printf ("  str_user_max     = %.2f s, str_sys_max     = %.2f s\n",
            str_user_max, str_sys_max);
      printf ("  beam_user_max    = %.2f s, beam_sys_max    = %.2f s\n",
            beam_user_max, beam_sys_max);
      printf ("  cfar_user_max    = %.2f s, cfar_sys_max    = %.2f s\n",
            cfar_user_max, cfar_sys_max);
      printf ("  report_user_max  = %.2f s, report_sys_max  = %.2f s\n",
            report_user_max, report_sys_max);

      printf ("\nWall clock timing -\n");
      printf ("  all_clock_time    = %f\n", all_clock_time);
      printf ("  disk_clock_time   = %f\n", disk_clock_time);
      printf ("  fft_clock_time    = %f\n", fft_clock_time);
      printf ("  index_clock_time  = %f\n", index_clock_time);
      printf ("  rewind_clock_time = %f\n", rewind_clock_time);
      printf ("  shift_clock_time  = %f\n", shift_clock_time);
      printf ("  str_clock_time    = %f\n", str_clock_time);
      printf ("  beam_clock_time   = %f\n", beam_clock_time);
      printf ("  cfar_clock_time   = %f\n", cfar_clock_time);
      printf ("  report_clock_time = %f\n", report_clock_time);
    }
```

```
  return;
}  /* main */
```

A.2 cell_avg_cfar.c

```
/*
 * cell_avg_cfar.c
 */

/*
 * This file contains the procedure cell_avg_cfar (), and is part of the
 * parallel HO-PD benchmark program written for the IBM SP2 by the STAP
 * benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the
 * ARPA Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * This procedure was largely untouched during our parallelization effort,
 * and therefore almost identical to the sequential version. Modifications
 * were made to perform the cell averaging and CFAR for only a slice of the
 * data cube, instead of the entire data cube (as was the case in the
 * sequential version of the program). As such, most, if not all, of the
 * comments in this procedure are taken directly from the sequential version
 * of this program.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>



/*
 * cell_avg_cfar ()
 *   inputs: threshold, dopplers, beams, rng, detect_cube
 *   outputs: target_report
 *
 *   threshold: when the power at a certain cell exceeds this threshold,
 *     we consider it as having a target
 *   dopplers: the number of dopplers in the input data
 *   beams: the number of beams (rows) in the input data
 *   rng: the number of range gates (columns) in the input matrix
 *   target_report: a 25-element target report matrix (each element consisting
 *     of four floating point numbers: RANG, BEAM, DOPPLER, and POWER), to
 *     output the 25 closest targets found in this slice of detect_cube
 *   detect_cube: input beam/doppler/rnggate matrix
 */

cell_avg_cfar (threshold, dopplers, beams, rng, target_report, detect_cube)
     float threshold;
     int dopplers;
     int beams;
     int rng;
     float target_report[][4];
     COMPLEX detect_cube[][DOP/NN][RNG];

{

/*
 * Variables: Most, if not all, of the variable comments were taken directly
```

```
 * from the sequential version of this program.
 *
 * sum: a holder for the cell sum
 * num_seg: number of range segments
 * rng_seg: number of range gates per segment
 * targets: holder for the number of targets to include in the target report
 * num_targets: the number of targets to include in the target report
 * N: holder for the number of range gates per range segment
 * guard_range: number of cells away from cell of interest
 * rseg, r, i: loop variables
 * beam, dop: loop variables
 * rng_start: address of the first range gate in each range segment
 * rng_end: address of the last range gate in each range segment
 * ncells: holder for cells: guard_range per range segment
 * taskid: the identifier for this task (added for parallel execution)
 */

  float sum;
  int num_seg = NUMSEG;
  int rng_seg = RNGSEG;
  int targets;
  int num_targets = 25;
  int N;
  int guard_range = 3;
  int rseg, r, i;
  int beam, dop;
  int rng_start;
  int rng_end;
  int ncells;

  extern int taskid;

/*
 * Begin function body: cell_avg_cfar ()
 *
 * Process cell averaging cfar by range segments: do cell averaging in ranges.
 */

  N = rng_seg;  /* Number of range cells per range segment */
  targets = 0;  /* Start with 0 targets in target report number */
  for (i = 0; i < num_targets; i++)
    {
      target_report[i][0] = 0.0;
      target_report[i][1] = 0.0;
      target_report[i][2] = 0.0;
      target_report[i][3] = 0.0;
    }

/*
 * Do the entire cell average CFAR algorithm starting from the first range
 * segment and continuing to the last range segment. This ensures getting
 * 25 least-range targets first, so the process can be stopped without
 * looking further into longer ranges.
 */

  for (rseg = 0; rseg < num_seg; rseg++)
    {  /* for (rseg...) */

/*
 * Get start and end range gates for each of the range segments: starts at
 * low ranges and increments to higher ranges.
 */

      rng_start = rseg * N;
      rng_end = rng_start + N - 1;

/*
 * 1st get the range cell power and store it in detect_cube[][][].real. For
 * each beam and each doppler, get the power for each range cell in the
```

```
 * current range segment.
 */

      for (beam = 0; beam < beams; beam++)
       {  /* for (beam...) */
         for (dop = 0; dop < dopplers; dop++)
           {  /* for (dop...) */
             for (r = rng_start ; r <= rng_end; r++)
              {  /* for (r...) */
                sum = detect_cube[beam][dop][r].real
                  * detect_cube[beam][dop][r].real
                    + detect_cube[beam][dop][r].imag
                     * detect_cube[beam][dop][r].imag;
                detect_cube[beam][dop][r].real = sum;
              }  /* for (r...) */
           }  /* for (dop...) */
       }  /* for (beam...) */

/*
 * Now, get the range cell's average power, using cell averaging CFAR described
 * above, and store it in detect_cube[][][].imag. For each beam and each
 * doppler, get the average power for each range cell in the current range
 * segment.
 */

      for (beam = 0; beam < beams; beam++)
       {  /* for (beam...) */
         for (dop = 0; dop < dopplers; dop++)
           {  /* for (dop...) */
             sum = 0.0;
             ncells = N - guard_range - 1;

/*
 * Do a summation loop for the first range cell at the start of a range
 * segment.
 */

             for (r = rng_start + guard_range + 1; r <= rng_end; r++)
              {
                sum += detect_cube[beam][dop][r].real;
              }

/*
 * The average power is the sum divided by the number of cells in the
 * summation.
 */

             detect_cube[beam][dop][rng_start].imag = sum / ncells;

/*
 * Now, perform loops until the guard band is fully involved in the data.
 */

             for (r = rng_start + 1; r <= rng_start + guard_range; r++)
              {  /* for (r...) */
                sum = sum - detect_cube[beam][dop][r + guard_range].real;
                --ncells;
                detect_cube[beam][dop][r].imag = sum / ncells;
              }  /* for (r...) */


/*
 * Now, perform loops until the guard band reaches the range segment border.
 */

             for (r = rng_start + guard_range + 1;
                 r <= rng_end - guard_range; r++)
              {
                sum = sum + detect_cube[beam][dop][r-guard_range-1].real
```

```
                - detect_cube[beam][dop][r + guard_range].real;
              detect_cube[beam][dop][r].imag = sum / ncells;
            }

/*
 * Now, perform loops to the end of the range segment.
 */

            for (r = rng_end - guard_range + 1; r <= rng_end; r++)
            {
              sum += detect_cube[beam][dop][r - guard_range - 1].real;
              ++ncells;
              detect_cube[beam][dop][r].imag = sum / ncells;
            }
          }  /* for (dop...) */
      }  /* for (beam...) */

/*
 * Compare the range cell power to the range cell average power. Start at
 * the minimum range and increase the range until 25 targets are found. Put
 * the target's RANGE, BEAM, DOPPLER, and POWER level in a matrix called
 * "target_report". Store all values as floating point numbers. Some can get
 * converted to integers on printout later as required.
 */

     for (r = rng_start ; r <= rng_end; r++)
     {  /* for (r...) */
       for (beam = 0; beam < beams; beam++)
         {  /* for (beam...) */
           for (dop = 0; dop < dopplers; dop++)
           {  /* for (dop...) */
             if (((detect_cube[beam][dop][r].real
                  - detect_cube[beam][dop][r].imag) >
                  threshold) && (targets <= num_targets))
               {  /* if (((...)))  */
                 target_report[targets][0] = (float) r;
                 target_report[targets][1] = (float) beam;
                 target_report[targets][2] = (float) dop
                  + taskid * PRI / NN;
                 target_report[targets][3]
                  = detect_cube[beam][dop][r].real;
                 targets += 1;
                 if (targets >= num_targets)
                  {
                    goto quit_report;
                  }
               }  /* if (((...))) */
           }  /* for (dop...) */
         }  /* for (beam...) */
     }  /* for (r...) */
   }  /* for (rseg..) */
 quit_report: ;
  return;
}
```

A.3 cmd_line.c

```
/*
 * cmd_line.c
 */

/*
 * This file contains the procedure cmd_line (), and is part of the parallel
 * HO-PD benchmark program written for the IBM SP2 by the STAP benchmark
```

```
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The procedure cmd_line () extracts the name of the files from which the
 * input data cube and the steering vector data should be loaded. The
 * function of this parallel version of cmd_line () is different from that
 * of the sequential version, because the sequential version also extracted
 * the number of iterations the program should run and some reporting
 * options.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>

/*
 * cmd_line ()
 *   inputs: argc, argv
 *   outputs: str_name, data_name
 *
 *   argc, argv: these are used to get data from the command line arguments
 *   str_name: a holder for the name of the input steering vectors file
 *   data_name: a holder for the name of the input data file
 */

cmd_line (argc, argv, str_name, data_name)
     int argc;
     char *argv[];
     char str_name[LINE_MAX];
     char data_name[LINE_MAX];

{

/*
 * Begin function body: cmd_line ()
 */

  strcpy (str_name, argv[1]);
  strcat (str_name, ".str");
  strcpy (data_name, argv[1]);
  strcat (data_name, ".dat");
  return;
}
```

A.4 compute_beams.c

```
/*
 * compute_beams.c
 */

/*
 * This file contains the procedure compute_beams (), and is part of the
 * parallel HO-PD benchmark program written for the IBM SP2 by the STAP
 * benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
```

```
 * on 10/22/93.
 *
 * The procedure compute_beams does beamforming per doppler. Its input is a
 * set of BEAM * NUMSEG number of weights for one doppler. It returns the
 * results in a new output cube consisting of BEAM beams by DOP dopplers by
 * RNG range gates. Nominally, BEAM, NUMSEG, DOP, and RNG = 2, 2, 128, and
 * 1250, respectively.
 *
 * This procedure was largely untouched during our parallelization effort,
 * and is therefore almost identical to the sequential version. Modifications
 * were made to perform this step on only a slice of the data cube, instead of
 * the entire data cube (as was the case in the sequential version of the
 * program). As such, most, if not all, of the comments in this procedure are
 * taken directly from the sequential version of this program.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>



/*
 * compute_beams ()
 *   inputs: dop, local_cube, weights, temp_mat
 *   outputs: detect_cube
 *
 *   dop: the doppler number for which the beams are computed
 *   local_cube: input data cube and dopplers after the FFTs
 *   weights: matrices holding the input set of weights
 *   detect_cube: detection output data cube
 *   temp_mat: temporary matrix holding three adjacent dopplers
 */
compute_beams (dop, local_cube, weights, detect_cube, temp_mat)
     int dop;
     COMPLEX local_cube[][RNG][EL];
     COMPLEX weights[][NUMSEG][DOF];
     COMPLEX detect_cube[][DOP / NN][RNG];
     COMPLEX temp_mat[][COLS];

{

/*
 * Variables
 *
 * rng_seg: number of range gates per segment
 * rngsamp: sample range gates, used in each segment
 * num_seg: number of range segments for the Cholesky
 * dofs: number of DOFs (rows) in the input data
 * rng: number of range gates in the input data
 * el: number of elements in the steering vectors
 * i, j, k, m, n: loop counters
 * dopplers: holder for the number of dopplers in data_cube
 * row: row number
 * col: column number
 * appl: column number for application data destination
 * apply: column number for application data source
 * beam: beam number to be formed in output data
 * seg: segment number
 * start_seg: column number for start range gate for a segment
 * doplr1, doplr2, doplr3: loop countgers for doppler number
 * xreal, ximag: storage for real and imaginary data for pointer use
 * real_ptr: pointer to next real data
 * imag_ptr: pointer to next imaginary data
 * sum: COMPLEX variable to hold the sum of complex data
 * pricnt: number of PRIs in the input data cube
 */
```

```
  int rng_seg = RNGSEG;
  int rngsamp = RNG_S;
  int num_seg = NUMSEG;
  int dofs = DOF;
  int rng = RNG;
  int el = EL;
  int i, j, k, m, n;
  int dopplers;
  int row;
  int col;
  int appl;
  int apply;
  int beam;
  int seg;
  int start_seg;
  int doplr1;
  int doplr2;
  int doplr3;
  float xreal[DOF];
  float ximag[DOF];
  float *real_ptr;
  float *imag_ptr;
  COMPLEX sum;
  extern int pricnt;

/*
 * Begin function body: compute_beams ()
 *
 * Perform beam forming for all dopplers. Determine the doppler numbers for
 * the doppler of interest, as well as the preceding and following dopplers.
 */

  doplr1 = dop - 1;
  doplr2 = dop;
  doplr3 = dop + 1;

/*
 * For each doppler, apply BEAM * NUMSEG set of weights to the data.
 */

  for (beam = 0; beam < BEAM; beam++)
    {  /* for (beam...) */
      for (seg = 0; seg < num_seg; seg++)
       {  /* for (seg...) */

/*
 * Apply the conjugate of the weight vector to all RNGSEG range gates for each
 * segment. Store the result in the output detection cube. If seg = 0 apply the
 * weights to the following segment, else for all other seg numbers apply the
 * weights to preceding segment.
 */

        if (seg == 0)
          {
            start_seg = rng_seg;
          }
        else
          {
            start_seg = (seg - 1) * rng_seg;
          }

/*
 * Temporary storage for the weight vector with pointers to speed multiply.
 */

        real_ptr = xreal;
        imag_ptr = ximag;
```

```
/*
 * Store the conjugate of the weight vector.
 */

        for (n = 0; n < dofs; n++)
          {
            *real_ptr++ = weights[beam][seg][n].real;
            *imag_ptr++ = -weights[beam][seg][n].imag;
          }

/*
 * Put application data elements in temp_mat. 1st EL rows.
 */

        for (k = 0; k < el; k++)
          {  /* for (k...) */
            row = k;
            for (col = 0; col < rng_seg; col++)
             {  /* for (col...) */
               appl = col + start_seg;
                                        /* Application region          */

/*
 * Get all RNGSEG range gates from one of NUMSEG segments.
 */

               temp_mat[row][col].real = local_cube[doplr1][appl][k].real;
               temp_mat[row][col].imag = local_cube[doplr1][appl][k].imag;
             }  /* for (col...) */
          }  /* for (k...) */

/*
 * 2nd EL rows.
 */

        for (k = 0; k < el; k++)
          {  /* for (k...) */
            row = el + k;
            for (col = 0; col < rng_seg; col++)
             {  /* for (col...) */
               appl = col + start_seg;
                                        /* Application region          */

/*
 * Get all RNGSEG range gates from one of NUMSEG segments.
 */

               temp_mat[row][col].real = local_cube[doplr2][appl][k].real;
               temp_mat[row][col].imag = local_cube[doplr2][appl][k].imag;
             }  /* for (col...) */
          }  /* for (k...) */

/*
 * 3rd EL rows.
 */

        for (k = 0; k < el; k++)
          {  /* for (k...) */
            row = 2 * el + k;
            for (col = 0; col < rng_seg; col++)
             {  /* for (col...) */
               appl = col + start_seg;
                                        /* Application region          */

/*
 * Get all RNGSEG range gates from one of NUMSEG segments.
 */

               temp_mat[row][col].real = local_cube[doplr3][appl][k].real;
```

```
            temp_mat[row][col].imag = local_cube[doplr3][appl][k].imag;
          }  /* for (col...) */
        }  /* for (k...) */

/*
 * Multiply the weight vector by all range gates of the application data.
 * Application data set at 1st RNGSEG cols of "temp_mat" matrix.
 */

      for (m = 0; m < rng_seg ; m++)
        {  /* for (m...) */
          sum.real = 0.0;
          sum.imag = 0.0;
          real_ptr = xreal;
          imag_ptr = ximag;
          for (row = 0; row < dofs; row++)
          {  /* for (row...) */
            sum.real += (temp_mat[row][m].real * *real_ptr -
                         temp_mat[row][m].imag * *imag_ptr);
            sum.imag += (temp_mat[row][m].imag * *real_ptr++ +
                         temp_mat[row][m].real * *imag_ptr++);
          }  /* for (row...) */

/*
 * Store the result into the detection data cube by col = range gate. Store the
 * result in the same range gates used for application region.
 */

          col = m + start_seg;
          detect_cube[beam][dop - 1][col].real = sum.real;
          detect_cube[beam][dop - 1][col].imag = sum.imag;
        }  /* for (m...) */
      }  /* for (seg...) */
    }  /* for (beam...) */
  return;
}
```

A.5 compute_weights.c

```
/*
 * compute_weights.c
 */

/*
 * This file contains the procedure compute_weights (), and is part of the
 * parallel HO-PD benchmark program written for the IBM SP2 by the STAP
 * benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The compute_weights () procedure does the weights computation prior to
 * beamforming. Its input is a dop and dopplers formed in the fft () routine.
 * Also, the modified space-time steering vectors are input to this routine.
 * It returns a set of weights to be used in the beam computation routine.
 *
 * For each input doppler number:
 *
 *   For each of the NUMSEG segments (of RNGSEG range gates each):
 *     a. For each doppler of interest, take two adjacent dopplers (doppler
 *        of interest and preceding and succeeding dopplers), making 3*EL =
 *        DOFS number of rows and RNG_S range gates (every other range gate,
```

```
 *        0, 2, 4, 6, etc. until RNG_S number of range gates). Nominally,
 *        RNG_S = 288.
 *     b. Do a Householder on all DOFs (rows), lower triangularizing all rows.
 *        Since we have NUMSEG segments of range gates, we do a Cholesky and
 *        a forback solve on a segment for later application of weights to one
 *        of the other segments adjacent to it. Starting range gate = 0 for
 *        Cholesky for the first segment. Starting range gate = NUMSEG * RNGSEG
 *        for Cholesky for the second or higher segment and apply weights to
 *        range gates starting at (NUMSEG-1) * RNGSEG segment (previous
 *        segment).
 *     c. Forward and back solve each DOF x DOF lower triangular matrix using
 *        modified space-time steering vectors DOF x 1. There will be RNGSEG
 *        separate sets of weights, per beam per doppler. Space-time steering
 *        vectors have already been formed composed of the concatenation of
 *        doppler-transformed spatial steering vectors for the doppler of
 *        interest and preceding and succeeding dopplers.
 *     d. Pass weights from (c.) above to caller to apply weights. Weights are
 *        calculated for each beam and each range segment and are passed in a
 *        matrix to be applied to data for each doppler.
 *
 * This procedure was largely untouched during our parallelization effort, and
 * is therefore almost identical to the sequential version. Modifications were
 * made to perform this step on only a slice of the data cube, instead of the
 * entire data cube (as was the case in the sequential version of the program).
 * As such, most, if not all, of the comments in this procedure are taken
 * directly from the sequential version of this program.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>



/*
 * compute_weights ()
 *   inputs: dop, local_cube, mod_str_vecs, temp_mat
 *   outputs: weights
 *
 *   dop: number of the doppler for weight computation
 *   local_cube: input data cube and dopplers after FFTs
 *   weights: weights matrix per doppler
 *   mod_str_vecs: matrices holding the modified steering vectors
 *   temp_mat: temporary matrix holding three adjacent dopplers
 */

compute_weights (dop, local_cube, weights, mod_str_vecs, temp_mat)
     int dop;
     COMPLEX local_cube[][RNG][EL];
     COMPLEX weights[][NUMSEG][DOF];
     COMPLEX mod_str_vecs[][DOP][DOF];
     COMPLEX temp_mat[][COLS];

{

/*
 * Variables
 *
 * rng_seg: number of range gates per segment
 * rngsamp: sample range gates used in each segment
 * num_seg: number of range segments for the Cholesky
 * dofs: numebr of DOFS (rows) in the input data
 * rng: number of range gates in the input data
 * el: number of elements in the steering vectors
 * num_rows: number of rows on which to apply the Cholesky
 * start_row: the row number on which to start applying the Cholesky
 * lower_triangular_rows: number of rows to lower triangularize
```

```
* i, j, k, m, n: loop counters
* dopplers: holder for the number of dopplers
* row: row number
* col: column number
* chol: column number for Cholesky data
* beam: beam number to be formed in the output data
* seg: segment number
* doplr1, doplr2, doplr3: loop counters for the doppler number
* xreal, ximag: storage for real and imaginary data for pointer use
* weight_vec: storage for weight solution vector
* mod_str_vec: storage for one modified steering vector
* make_t: flag: 0 = T matrix is not made and weight vector is returned to
*   caller
* pricnt: number of PRIs in the input local_cube
* taskid: identifier number for this task
*/

  int rng_seg = RNGSEG;
  int rngsamp = RNG_S;
  int num_seg = NUMSEG;
  int dofs = DOF;
  int rng = RNG;
  int el = EL;
  int num_rows;
  int start_row;
  int lower_triangular_rows;
  int i, j, k, m, n;
  int dopplers;
  int row;
  int col;
  int chol;
  int beam;
  int seg;
  int doplr1;
  int doplr2;
  int doplr3;
  float xreal[DOF];
  float ximag[DOF];
  COMPLEX weight_vec[DOF];
  COMPLEX mod_str_vec[1][DOF];
  int make_t;
  extern pricnt;
  extern void house();
  extern void forback();
  extern taskid;

  int dopxu;

/*
 * Begin function body: compute_weights ()
 *
 * Set make_t = 0 so we don't form the T matrix in the forback () routine.
 */

  make_t = 0;

/*
 * Because of the way the data cube was redistributed (using the index and two
 * circular shifts), each node has all the dopplers it needs to do its
 * computation. For each doppler, form a DOF (3 * EL) rows by RNG_S columns
 * matrix using the elements for every other range gate from the doppler of
 * interest and previous and following dopplers.
 */

  doplr1 = dop - 1;
  doplr2 = dop;
  doplr3 = dop + 1;

  dopxu = (taskid * PRI / NN) + dop - 1;
```

```
/*
 * Perform weight computations for all beams for all segments. Perform a
 * Householder tranform and a forward and back solution for the weights for
 * each segment.
 */

  for (seg = 0; seg < num_seg; seg++)
    {  /* for (seg...) */

/*
 * 1st, Put Cholesky data elements in temp_mat to do householder. 1st EL rows
 */

      for (k = 0; k < el; k++)
       {  /* for (k...) */
         row = k;
         for (col = 0; col < rngsamp; col++)
           {  /* for (col...) */
             chol = 2 * col + seg * rng_seg;
                                        /* Cholesky data                */

/*
 * Get every other range gate from either 1st or 2nd segment.
 */

             temp_mat[row][col].real = local_cube[doplr1][chol][k].real;
             temp_mat[row][col].imag = local_cube[doplr1][chol][k].imag;
           }  /* for (col...) */
       }  /* for (k...) */

/*
 * 2nd EL rows.
 */

      for (k = 0; k < el; k++)
       {  /* for (k...) */
         row = el + k;
         for (col = 0; col < rngsamp; col++)
           {  /* for (col...) */
             chol = 2 * col + seg * rng_seg;
                                        /* Cholesky data                */

/*
 * Get every other range gate from either the 1st or 2nd segment.
 */

             temp_mat[row][col].real = local_cube[doplr2][chol][k].real;
             temp_mat[row][col].imag = local_cube[doplr2][chol][k].imag;
           }  /* for (col...) */
       }  /* for (k...) */

/*
 * 3rd EL rows.
 */

      for (k = 0; k < el; k++)
       {  /* for (k...) */
         row = 2 * el + k;
         for (col = 0; col < rngsamp; col++)
           {  /* for (col...) */
             chol = 2 * col + seg * rng_seg;
                                        /* Cholesky data                */

/*
 * Get every other range gate from either the 1st or 2nd segment.
 */

             temp_mat[row][col].real = local_cube[doplr3][chol][k].real;
```

```
            temp_mat[row][col].imag = local_cube[doplr3][chol][k].imag;
          }  /* for (col...) */
        }  /* for (k...) */

/*
 * Perform a Householder transform to lower triangularize all rows.
 */

      num_rows = dofs;
      lower_triangular_rows = dofs;
      start_row = 0;                    /* start householder on 1st row */

/*
 * Call the Householder routine.
 */

      house (num_rows, rngsamp, lower_triangular_rows, start_row, temp_mat);

/*
 * temp_mat matrix now has a lower triangular matrix as the 1st MxM rows. M
 * nominally = 144 = DOF or 3 * EL.
 */

      for (beam = 0; beam < BEAM; beam++)
       {  /* for (beams...) */

/*
 * Move the appropriate steering vector into a temporary holding matrix.
 */

         for (n = 0; n < dofs; n++)
           {
             mod_str_vec[0][n].real = mod_str_vecs[beam][dopxu][n].real;
             mod_str_vec[0][n].imag = mod_str_vecs[beam][dopxu][n].imag;
           }

/*
 * Call forback to solve weight vector with modified steer vector.
 */

         forback (num_rows, mod_str_vec, weight_vec, make_t, temp_mat);

/*
 * Store weights into the weights matrix.
 */

         for (n = 0; n < dofs; n++)
           {
             weights[beam][seg][n].real = weight_vec[n].real;
             weights[beam][seg][n].imag = weight_vec[n].imag;
           }
       }  /* for (beams...) */
    }  /* for (seg...) */
  return;
}
```

A.6 fft.c

```
/*
 * fft.c
 */

/*
 * This file contains the procedures fft () and bit_reverse (), and is part
```

```
 * of the parallel HO-PD benchmark program written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The procedure fft () implements an n-point in-place decimation-in-time
 * FFT of complex vector "data" using the n/2 complex twiddle factors in
 * "w_common". The implementation used in this procedure is a hybrid between
 * the implementation in the original sequential version of this program and
 * the implementation suggested by MHPCC. This modification was made to
 * improve the performance of the FFT on the SP2.
 *
 * The procedure bit_reverse () implements a simple (but somewhat inefficient)
 * bit reversal.
 */

#include "defs.h"

/*
 * fft ()
 *   inputs: data, w_common, n, logn
 *   outputs: data
 */

void fft (data, w_common, n, logn)
     COMPLEX *data, *w_common;
     int n, logn;

{  /* fft */
  int incrvec, i0, i1, i2, nx, t1, t2, t3;
  float f0, f1;
  void bit_reverse();

/*
 * Begin function body: fft ()
 *
 * Bit-reverse the input vector.
 */

  (void) bit_reverse (data, n);

/*
 * Do the first log n - 1 stages of the FFT.
 */

  i2 = logn;
  for (incrvec = 2; incrvec < n; incrvec <<= 1)
    {  /* for (incrvec...) */
      i2--;
      for (i0 = 0; i0 < incrvec >> 1; i0++)
       {  /* for (i0...) */
         for (i1 = 0; i1 < n; i1 += incrvec)
           {  /* for (i1...) */
             t1 = i0 + i1 + incrvec / 2;
             t2 = i0 << i2;
             t3 = i0 + i1;
             f0 = data[t1].real * w_common[t2].real -
              data[t1].imag * w_common[t2].imag;
             f1 = data[t1].real * w_common[t2].imag +
              data[t1].imag * w_common[t2].real;
             data[t1].real = data[t3].real - f0;
             data[t1].imag = data[t3].imag - f1;
             data[t3].real = data[t3].real + f0;
             data[t3].imag = data[t3].imag + f1;
           }  /* for (i1...) */
       }  /* for (i0...) */
```

```
    }  /* for (incrvec...) */

/*
 * Do the last stage of the FFT.
 */

  for (i0 = 0; i0 < n / 2; i0++)
    {  /* for (i0...) */
      t1 = i0 + n / 2;
      f0 = data[t1].real * w_common[i0].real -
       data[t1].imag * w_common[i0].imag;
      f1 = data[t1].real * w_common[i0].imag +
       data[t1].imag * w_common[i0].real;
      data[t1].real = data[i0].real - f0;
      data[t1].imag = data[i0].imag - f1;
      data[i0].real = data[i0].real + f0;
      data[i0].imag = data[i0].imag + f1;
    }  /* for (i0...) */
}  /* fft */

/*
 * bit_reverse ()
 *   inputs: a, n
 *   outputs: a
 */

void bit_reverse (a, n)
     COMPLEX *a;
     int n;

{
  int i, j, k;

/*
 * Begin function body: bit_reverse ()
 */

  j = 0;
  for (i = 0; i < n - 2; i++)
    {
      if (i < j)
       SWAP(a[j], a[i]);
      k = n >> 1;
      while (k <= j)
       {
         j -= k;
         k >>= 1;
       }
      j += k;
    }
}
```

A.7 fft_STAP.c

```
/*
 * fft_STAP.c
 */

/*
 * This file contains the procedure fft_STAP (), and is part of the parallel
 * HO-PD benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
```

```
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The procedure fft_STAP () performs an FFT along the PRI dimension for each
 * column of data (there are a total of RNG*EL such columns) by calling the
 * procedure fft ().
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>



/*
 * fft_STAP ()
 *   inputs: fft_out
 *   outputs: fft_out
 *
 *   fft_out the FFT output data cube
 */

fft_STAP (fft_out)
     COMPLEX fft_out[][RNG / NN][EL];

{

/*
 * Variables
 *
 * pricnt: pricnt globally defined in main
 * fft (): a pointer to the FFT function (which performs a radix 2
 *   calculation)
 * el: the maximum number of elements in each PRI
 * beams: the maximum number of beams
 * rng: the maximum number of range gates in each PRI, in this data cube slice
 *   (modified for parallel execution)
 * i, j, k: loop counters
 * xrealptr, ximagptr: not used
 * logpoints: log base 2 of the number of points in the FFT
 * pix2, pi: 2*pi and pi
 * x: storage for the temporary FFT vector
 * w: storage for the twiddle factors table
 */

  extern int pricnt;
  extern void fft();
  int el = EL;
  int beams = BEAM;
  int rng = RNG / NN;
  int i, j, k;
  float *xrealptr;
  float *ximagptr;
  int logpoints;
  float pi, pix2;
  static COMPLEX x[PRI];
  static COMPLEX w[PRI];

/*
 * Begin function body: fft_STAP ()
 *
 * Generate twiddle factors table w.
 */

  logpoints = log2 ((float) pricnt) + 0.1;
  pi = 3.14159265358979;
```

```
  pix2 = 2.0 * pi;

  for (i = 0; i < PRI; i++)
    {
      w[i].imag = -sin (pix2 * (float) i / (float) PRI);
      w[i].real = cos (pix2 * (float) i / (float) PRI);
    }

/*
 * Perform one FFT for each rng and el.
 */

  for (i = 0; i < rng; i++)
    {  /* for (i...) */
      for (k = 0; k < el; k++)
       {  /* for (k...) */

/*
 * Move all the pri's into a vector.
 */

         for (j = 0; j < pricnt; j++)
           {
             x[j].real = fft_out[j][i][k].real;
             x[j].imag = fft_out[j][i][k].imag;
           }

/*
 * Call the FFT routine.
 */

         fft (x, w, pricnt, logpoints);

/*
 * Move FFT'ed data back into fft_out.
 */

         for (j = 0; j < pricnt; j++)
           {
             fft_out[j][i][k].real = x[j].real;
             fft_out[j][i][k].imag = x[j].imag;
           }
       }  /* for (k...) */
    }  /* for (i...) */
  return;
}
```

A.8 forback.c

```
/*
 * forback.c
 */

/*
 * This file contains the procedure forback (), and is part of the parallel
 * HO-PD benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The procedure forback () performs a forward and back substitution on an
```

```
 * input temp_mat array, using the steering vectors, str_vecs, and
 * normalizes the solution returned in the vector "weight_vec".
 *
 * If the input variable "make_t" = 1, then the T matrix is generated in
 * this routine by doing BEAM forward and back solution vectors. Nominally,
 * BEAM = 32.
 *
 * The conjugate transpose of each weight vector, to get the Hermitian, is
 * put into the T matrix as row vectors. The input, temp_mat, is a square
 * lower triangular array, with the number of rows and columns = num_rows.
 *
 * This routine requires 5 input parameters as follows:
 *   num_rows: the number of COMPLEX elements, M, in temp_mat
 *   str_vecs: a pointer to the start of the steering vector array
 *   weight_vec: a pointer to the start of the COMPLEX weight vector
 *   make_t: 1 = make T matrix; 0 = don't make T matrix but pass back the
 *     weight vector
 *   temp_mat[][]: a temporary matrix holding various data
 *
 * This procedure was untouched during our parallelization effort, and
 * therefore is virtually identical to the sequential version. Also, most,
 * if not all, of the comments were taken verbatim from the sequential
 * version.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"

/*
 * forback ()
 *   inputs: num_rows, str_vecs, weight_vec, make_t, temp_mat
 *   outputs: weight_vec
 *
 *   num_rows: the number of elements, M, in the input data
 *   str_vecs: a pointer to the start of the steering vector array
 *   weight_vec: a pointer to the start of the COMPLEX weight vector
 *   make_t: 1 = make T matrix; 0 = don't make T matrix but pass back the
 *     weight vector
 *   temp_mat: MAX temprary matrix holding various data
 */

forback (num_rows, str_vecs, weight_vec, make_t, temp_mat)
     int num_rows;
     COMPLEX str_vecs[][DOF];
     COMPLEX weight_vec[];
     int make_t;
     COMPLEX temp_mat[][COLS];

{

/*
 * Variables
 *
 * i, j, k: loop counters
 * last: the last row or element in the matrices or vectors
 * beams: loop counter
 * num_beams: loop counter
 * sum_sq: a holder for the sum squared of the elements in a row
 * sum: a holder for the sum of the COMPLEX elements in a row
 * temp: a holder for temporary storage of a COMPLEX element
 * abs_mag: the absolute magnitude of a complex element
 * wt_factor: a holder for the weight normalization factor, according to
 *   Adaptive Matched Filter Normalization
 * steer_vec: one steering vector with DOF COMPLEX elements
 * vec: a holder for the complex solution intermediate vector
 * tmp_mat: a pointer to the start of the COMPLEX temp matrix
 */
```

```
  int i, j, k;
  int last;
  int beams;
  int num_beams;
  float sum_sq;
  COMPLEX sum;
  COMPLEX temp;
  float abs_mag;
  float wt_factor;
  COMPLEX steer_vec[DOF];
  COMPLEX vec[DOF];
  COMPLEX tmp_mat[DOF][DOF];

/*
 * Begin function body: forback ()
 *
 * The temp_mat matrix contains a lower triangular COMPLEX matrix as the
 * first MxM rows. Do a forward back solution BEAM times with a different
 * steering vector. If make_t = 1, then make the T matrix.
 */

  if (make_t)
    {
      num_beams = num_rows;  /* Do M forward back solution vectors. Put */
                             /* them else into T matrix. */
    }

  else
    {
      num_beams = 1;  /* Do only 1 solution weight vector */
    }

  for (beams = 0; beams < num_beams; beams++)
    {  /* for (beams...) */
      for (j = 0; j < num_rows; j++)
       {
         steer_vec[j].real = str_vecs[beams][j].real;
         steer_vec[j].imag = str_vecs[beams][j].imag;
       }

/*
 * Step 1: Do forward elimination. Also, get the weight factor = square root
 * of the sum squared of the solution vector. Used to divide back substitution
 * solution to get the weight vector. Divide the first element of the COMPLEX
 * steering vector by the first COMPLEX diagonal to get the first element of
 * the COMPLEX solution vector. First, get the absolute magnitude of the first
 * lower triangular diagonal.
 */

      abs_mag = temp_mat[0][0].real * temp_mat[0][0].real
                + temp_mat[0][0].imag * temp_mat[0][0].imag;

/*
 * Solve for the first element of the solution vector.
 */

      vec[0].real = (temp_mat[0][0].real * steer_vec[0].real +
                     temp_mat[0][0].imag * steer_vec[0].imag) / abs_mag;
      vec[0].imag = (temp_mat[0][0].real * steer_vec[0].imag -
                     temp_mat[0][0].imag * steer_vec[0].real) / abs_mag;

/*
 * Start summing the square of the solution vector.
 */

      sum_sq = vec[0].real * vec[0].real + vec[0].imag * vec[0].imag;

/*
 * Now solve for the remaining elements of the solution vector.
```

```
 */

      for (i = 1; i < num_rows; i++)
        {  /* for (i...) */
         sum.real = 0.0;
         sum.imag = 0.0;
         for (k = 0; k < i; k++)
           {  /* for (k...) */
             sum.real += (temp_mat[i][k].real * vec[k].real -
                          temp_mat[i][k].imag * vec[k].imag);
             sum.imag += (temp_mat[i][k].imag * vec[k].real +
                          temp_mat[i][k].real * vec[k].imag);
           }  /* for (k...) */

/*
 * Now subtract the sum from the next element of the steering vector.
 */

         temp.real = steer_vec[i].real - sum.real;
         temp.imag = steer_vec[i].imag - sum.imag;

/*
 * Get the absolute magnitude of the next diagonal.
 */

         abs_mag = temp_mat[i][i].real * temp_mat[i][i].real
                     + temp_mat[i][i].imag * temp_mat[i][i].imag;

/*
 * Solve for the next element of the solution vector.
 */

         vec[i].real = (temp_mat[i][i].real * temp.real +
                        temp_mat[i][i].imag * temp.imag) / abs_mag;
         vec[i].imag = (temp_mat[i][i].real * temp.imag -
                        temp_mat[i][i].imag * temp.real) / abs_mag;

/*
 * Sum the square of the solution vector.
 */

         sum_sq += (vec[i].real * vec[i].real + vec[i].imag * vec[i].imag);
        }  /* for (i...) */
      wt_factor = sqrt ((double) sum_sq);

/*
 * Step 2: Take the conjugate transpose of the lower triangular matrix to
 * form an upper triangular matrix.
 */

      for (i = 0; i < num_rows; i++)
       {  /* for (i...) */
         for (j = 0; j < num_rows; j++)
            {  /* for (j...) */
             tmp_mat[i][j].real = temp_mat[j][i].real;
             tmp_mat[i][j].imag = - temp_mat[j][i].imag;
            }  /* for (j...) */
       }   /* for (i...) */

/*
 * Step 3: Do a back substitution.
 */

      last = num_rows - 1;

/*
 * Get the absolute magnitude of the last upper triangular diagonal.
 */
```

```
      abs_mag = tmp_mat[last][last].real * tmp_mat[last][last].real
                + tmp_mat[last][last].imag * tmp_mat[last][last].imag;

/*
 * Solve for the last element of the weight solution vector.
 */

      weight_vec[last].real = (tmp_mat[last][last].real * vec[last].real
                             + tmp_mat[last][last].imag * vec[last].imag)
                              / abs_mag;
      weight_vec[last].imag = (tmp_mat[last][last].real * vec[last].imag
                             - tmp_mat[last][last].imag * vec[last].real)
                              / abs_mag;

/*
 * Now solve for the remaining elements of the weight solution vector from
 * the next to last element up to the first element.
 */

      for (i = last - 1; i >= 0; i--)
        {  /* for (i...) */
          sum.real = 0.0;
          sum.imag = 0.0;
          for (k = i + 1; k <= last; k++)
            {  /* for (k...) */
              sum.real += (tmp_mat[i][k].real * weight_vec[k].real -
                           tmp_mat[i][k].imag * weight_vec[k].imag);
              sum.imag += (tmp_mat[i][k].imag * weight_vec[k].real +
                           tmp_mat[i][k].real * weight_vec[k].imag);
            }  /* for (k...) */

/*
 * Subtract the sum from the next element up of the forward solution vector.
 */

          temp.real = vec[i].real - sum.real;
          temp.imag = vec[i].imag - sum.imag;

/*
 * Get the absolute magnitude of the next diagonal up.
 */

          abs_mag = tmp_mat[i][i].real * tmp_mat[i][i].real
                    + tmp_mat[i][i].imag * tmp_mat[i][i].imag;

/*
 * Solve for the next element up of the weight solution vector.
 */

          weight_vec[i].real = (tmp_mat[i][i].real * temp.real +
                                tmp_mat[i][i].imag * temp.imag) / abs_mag;
          weight_vec[i].imag = (tmp_mat[i][i].real * temp.imag -
                                tmp_mat[i][i].imag * temp.real) / abs_mag;
        }  /* for (i...) */

/*
 * Step 4: Divide the solution weight_vector by the weight factor.
 */

       for (i = 0; i < num_rows; i++)
        {
          weight_vec[i].real /= wt_factor;
          weight_vec[i].imag /= wt_factor;
        }

#ifdef APT

/*
 * If make_t = 1, make the T matrix.
```

```
 */

      if (make_t)
       {  /* if (make_t) */

/*
 * Conjugate transpose the weight vector to get the Hermitian.  Put each
 * weight vector into the T matrix as row vectors.
 */

         for (j = 0; j < num_rows; j++)
           {
             t_matrix[beams][j].real = weight_vec[j].real;
             t_matrix[beams][j].imag = - weight_vec[j].imag;
           }
       }  /* if (make_t) */
#endif

    }  /* for (beams...) */
  return;
}
```

A.9 form_beams.c

```
/*
 * form_beams.c
 */

/*
 * This file contains the procedure form_beams (), and is part of the parallel
 * HO-PD benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The procedure form_beams () performs the beamforming by calling the
 * compute_weights () and compute_beams () procedures.
 */


#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>
extern int taskid, numtask, allgrp;



/*
 * form_beams ()
 *   inputs: dopplers, local_cube, mod_str_vecs
 *   outputs: detect_cube
 *
 *   dopplers: number of PRIs in the input data
 *   local_cube: input data cube and dopplers after FFTs
 *   detect_cube: detection data cube
 *   mod_str_vecs: matrices holding modified steering vectors
 */

form_beams (dopplers, local_cube, detect_cube, mod_str_vecs)
```

```
    int dopplers;
    COMPLEX local_cube[][RNG][EL];
    COMPLEX detect_cube[][DOP / NN][RNG];
    COMPLEX mod_str_vecs[][DOP][DOF];

{

/*
 * Variables
 *
 * temp_mat: a temporary matrix holding three adjacent dopplers
 * weights: weights matrix per doppler to apply to range segment per beam
 * dop: loop counter for doppler number
 */

  static COMPLEX temp_mat[DOF][COLS];
  static COMPLEX weights[BEAM][NUMSEG][DOF];
  int dop;
  extern void compute_weights();
  extern void compute_beams();

/*
 * Begin function body: form_beams ().
 *
 * Perform beamforming for all dopplers.
 */

  for (dop = 1; dop <= dopplers; dop++)
    {
      compute_weights (dop, local_cube, weights, mod_str_vecs, temp_mat);
      compute_beams (dop, local_cube, weights, detect_cube, temp_mat);
    }
  return;
}
```

A.10 form_str_vecs.c

```
/*
 * form_str_vecs.c
 */

/*
 * This file contains the procedure form_str_vecs (), and is part of the
 * parallel HO-PD benchmark program written for the IBM SP2 by the STAP
 * benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>



/*
 * form_str_vecs ()
 *   inputs: str_vecs
 *   outputs: mod_str_vecs
```

```
 *
 *    str_vecs: storage for complex spatial steering vectors
 *    mod_str_vecs: modified steering vectors matrices
 */

form_str_vecs (str_vecs, mod_str_vecs)
     COMPLEX str_vecs[][EL];
     COMPLEX mod_str_vecs[][DOP][DOF];

{

/*
 * Variables
 *
 * el: number of elements in each PRI
 * element: number of the element
 * beams: number of beams
 * dops: number of dopplers after the FFT
 * i, j, k: loop counters
 */

  int el = EL;
  int element;
  int beams = BEAM;
  int dops = DOP;
  int i, j, k;

/*
 * Begin function body: form_str_vecs ()
 *
 * Special case steering vectors.
 */

  for (i = 0; i < beams; i++)
    {  /* for (i...) */
      for (j = 0; j < dops; j++)
       {  /* for (j...) */

/*
 * 1st, put EL number 0,0 in special case steering vectors.
 */

          for (k = 0; k < el; k++)
            {  /* for (k...) */
              mod_str_vecs[i][j][k].real = 0.0;
              mod_str_vecs[i][j][k].imag = 0.0;
            }  /* for (k...) */

/*
 * 2nd, put EL elements length spacial steering vector in special case
 * steering vectors.
 */

          for (k = 0; k < el; k++)
            {  /* for (k...) */
              element = k + el;
              mod_str_vecs[i][j][element].real = str_vecs[i][k].real;
              mod_str_vecs[i][j][element].imag = str_vecs[i][k].imag;
            }  /* for (k...) */

/*
 * 3rd, put EL number 0,0 in special case steering vectors.
 */

          for (k = 0; k < el; k++)
            {  /* for (k...) */
              element = k + 2 * el;
              mod_str_vecs[i][j][element].real = 0.0;
              mod_str_vecs[i][j][element].imag = 0.0;
```

```
            }  /* for (k...) */
        }  /* for (j...) */
    }  /* for (i...) */
  return;
}
```

A.11 house.c

```
/*
 * house.c
 */

/*
 * This file contains the procedure house (), and is part of the parallel
 * HO-PD benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The procedure house () performs the Householder transform, in place, on
 * an N by M complex imput matrix, where M >= N. It returns the results in
 * the same location as the input data.
 *
 * This routine requries 5 input parameters as follows:
 *   num_rows: number of elements in the temp_mat
 *   num_cols: number of range gates in the temp_mat
 *   lower_triangular_rows: the number of rows in the output temp_mat that
 *     have been lower triangularized
 *   start_row: the number of the row on which to start the Householder
 *   temp_mat[][]: a temporary matrix holding various data
 *
 * This procedure was untouched during our parallelization effort, and
 * therefore is virtually identical to the sequential version. Also, most,
 * if not all, of the comments were taken verbatim from the sequential
 * version.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"

/*
 * house ()
 *   inputs: num_rows, num_cols, lower_triangular_rows, start_row, temp_mat
 *   outputs: temp_mat
 *
 *   num_rows: number of elements, N, in the input data
 *   num_cols: number of range gates, M, in the input data
 *   lower_triangular_rows: the number of rows to be lower triangularized
 *   start_row: the row number of which to start the Householder
 *   temp_mat: temporary matrix holding various data
 */

house (num_rows, num_cols, lower_triangular_rows, start_row, temp_mat)
     int num_rows;
     int num_cols;
     int lower_triangular_rows;
     int start_row;
     COMPLEX temp_mat[][COLS];

{
```

```
/*
 * Variables
 *
 * i, j, k: loop counters
 * rtemp: a holder for temporary scalar data
 * x_square: a holder for the absolute square of complex variables
 * xmax_sq: a holder for the maximum of the complex absolute of variables
 * vec: a holder for the maximum complex vector 2 * num_cols max
 * sigma: a holder for a complex variable used in the Householder
 * gscal: a holder for a complex variable used in the Householder
 * alpha: a holder for a scalar variable used in the Householder
 * beta: a holder for a scalar variable used in the Householder
 */

  int i, j, k;
  float rtemp;
  float x_square;
  float xmax_sq;
  COMPLEX vec[2*COLS];
  COMPLEX sigma;
  COMPLEX gscal;
  float alpha;
  float beta;

/*
 * Begin function body: house ()
 *
 * Loop through temp_mat for number of rows = lower_triangular_rows. Start at
 * the row number indicated by the start_row input variable.
 */

  for (i = start_row; i < lower_triangular_rows; i++)
    {  /* for (i...) */

/*
 * Step 1: Find the maximum absolute element for each row of temp_mat,
 * starting at the diagonal element of each row.
 */

      xmax_sq = 0.0;
      for (j = i; j < num_cols; j++)
       {  /* for (j...) */
         x_square = temp_mat[i][j].real * temp_mat[i][j].real
                  + temp_mat[i][j].imag * temp_mat[i][j].imag;
         if (xmax_sq < x_square)
           {
             xmax_sq = x_square;
           }
       }  /* for (j...) */

/*
 * Step 2: Normalize the row by the maximum value and generate the complex
 * transpose vector of the row in order to calculate alpha = square root of
 * the sum square of all the elements in the row.
 */

      xmax_sq = (float) sqrt ((double) xmax_sq);
      alpha = 0.0;
      for (j = i; j < num_cols; j++)
       {  /* for (j...) */
         vec[j].real = temp_mat[i][j].real / xmax_sq;
         vec[j].imag = - temp_mat[i][j].imag / xmax_sq;
         alpha += (vec[j].real * vec[j].real + vec[j].imag * vec[j].imag);
       }  /* for (j...) */

      alpha = (float) sqrt ((double) alpha);

/*
```

```
 * Step 3: Find beta = 2 / (b (transpose) * b). Find sigma of the relevant
 * element = x(i) / |x(i)|.
 */

      rtemp = vec[i].real * vec[i].real + vec[i].imag * vec[i].imag;
      rtemp = (float) sqrt ((double) rtemp);
      beta = 1.0 / (alpha * (alpha + rtemp));
      if (rtemp >= 1.0E-16)
       {
         sigma.real = vec[i].real / rtemp;
         sigma.imag = vec[i].imag / rtemp;
       }
      else
       {
         sigma.real = 1.0;
         sigma.imag = 0.0;
       }

/*
 * Step 4: Calculate the vector operator for the relevent element.
 */

      vec[i].real += sigma.real * alpha;
      vec[i].imag += sigma.imag * alpha;

/*
 * Step 5: Apply the Householder vector to all the rows of temp_mat.
 */

      for (k = i; k < num_rows; k++)
       {  /* for (k...) */

/*
 * Find the scalar for finding g.
 */

        gscal.real = 0.0;
        gscal.imag = 0.0;

        for (j = i; j < num_cols; j++)
          {  /* for (j...) */
            gscal.real += (temp_mat[k][j].real * vec[j].real -
                           temp_mat[k][j].imag * vec[j].imag);
            gscal.imag += (temp_mat[k][j].real * vec[j].imag +
                           temp_mat[k][j].imag * vec[j].real);
           }  /* for (j...) */
        gscal.real *= beta;
        gscal.imag *= beta;

/*
 * Modify only the necessary elements of the temp_mat, subtracting gscal
 * * conjg (vec) from temp_mat elements.
 */

        for (j = i; j < num_cols; j++)
          {  /* for (j...) */
             temp_mat[k][j].real -= (gscal.real * vec[j].real +
                                     gscal.imag * vec[j].imag);
             temp_mat[k][j].imag -= (gscal.imag * vec[j].real -
                                     gscal.real * vec[j].imag);
           }  /*  for (j...) */
      }  /* for (k...) */
    }  /* for (i...) */
  return;
}
```

A.12 read_input_STAP.c

```
/*
 * read_input_STAP.c
 */

/*
 * This file contains the procedure read_input_STAP (), and is part of the
 * parallel HO-PD benchmark program written for the IBM SP2 by the STAP
 * benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential HO-PD benchmark program was originally written by Tony Adams
 * on 10/22/93.
 *
 * The procedure read_input_STAP () reads the input data files (containing
 * the data cube and the steering vectors).
 *
 * In this parallel version of read_input_STAP (), each task reads its portion
 * of the data cube in from the data file simultaneously. In order to improve
 * disk performance, the entire data cube slice is read from disk, then
 * converted from the packed binary integer format to the floating point
 * number format.
 *
 * Each complex number is stored on disk as a packed 32-bit binary integer.
 * The 16 LSBs are the real portion of the number, and the 16 MSBs are the
 * imaginary portion of the number.
 *
 * The steering vector file contains the number of PRIs in the input data,
 * the target power threshold, and the steering vectors. The data is stored
 * in ASCII format, and the complex steering vector numbers are stored as
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>

#ifdef DBLE
char *fmt = "%lf";
#else
char *fmt = "%f";
#endif

extern int taskid, numtask, allgrp;



/*
 * read_input_STAP ()
 *   inputs: data_name, str_name
 *   outputs: str_vecs, fft_out
 *
 *   data_name: name of the file containing the input data cube
 *   str_name: name of the file containing the steering vectors
 *   str_vecs: the steering vectors
 *   fft_out: this task's slice of the data cube
 */

read_input_STAP (data_name, str_name, str_vecs, fft_out)
     char data_name[];
     char str_name[];
     COMPLEX str_vecs[][EL];
     COMPLEX fft_out[][RNG / NN][EL];
```

```
{

/*
 * Variables
 *
 * el: the maximum number of elements in each PRI
 * beams: the maximum number of beams
 * rng: the maximum number of range gates
 * dopplers: number of dopplers in the input data
 * temp: a buffer for binary input integer data
 * temp1, temp2: holders for integer data
 * i, j, k: loop counters
 * fopen (): a pointer to the file open function
 * f_str, f_dat: pointers to the input files
 * pricnt: the pricnt variable globally defined in main ()
 * threshold: the threshold variable globally defined in main ()
 * taskid: the identifier for this task
 * local_int_cube: the local portion of the data cube
 */

  int el = EL;
  int beams = BEAM;
  int rng = RNG / NN;
  int dopplers;
  unsigned int temp[1];
  long int temp1, temp2;
  int i, j, k;
  FILE *fopen();
  FILE *f_str, *f_dat;
  extern int pricnt;
  extern float threshold;
  extern taskid;

  unsigned int local_int_cube[RNG / NN][PRI][EL];

/*
 * Begin function body: read_input_STAP ()
 *
 * Every task: load the steering vectors file.
 */

  if ((f_str = fopen (str_name, "r")) == NULL)
    {
      printf ("Error - task %d unable to open steering vector file.\n",
              taskid);
      exit (-1);
    }

/*
 * The first item in the steering vector file is the number of PRIs. The
 * second item in the steering vector file is the target detection threshold.
 */

  fscanf (f_str, "%d", &pricnt);
  fscanf (f_str, fmt, &threshold);

/*
 * Read in the rest of the steering vector file.
 */

  for (i = 0; i < beams; i++)
    {
      for (j = 0; j < el; j++)
       {
         fscanf (f_str, fmt, &str_vecs[i][j].real);
         fscanf (f_str, fmt, &str_vecs[i][j].imag);
       }
    }
```

```
  fclose (f_str);

/*
 * Read in the data file in parallel.
 */

  if ((f_dat = fopen (data_name, "r")) == NULL)
    {
      printf ("Error - task %d unable to open data file.\n", taskid);
      exit (-1);
    }

  fseek (f_dat, taskid * rng * pricnt * el * sizeof (unsigned int), 0);
  fread (local_int_cube, sizeof (unsigned int), pricnt * rng * el, f_dat);
  fclose (f_dat);

/*
 * Convert data from unsigned int format to floating point format.
 */

  for (i = 0; i < rng; i++)
    { /* for (i...) */
      for (j = 0; j < pricnt; j++)
       {  /* for (j...) */
         for (k = 0; k < el; k++)
           {  /* for (k...) */
             temp[0] = local_int_cube[i][j][k];
             temp1 = 0x0000FFFF & temp[0];
             temp1 = (temp1 & 0x00008000) ? temp1 | 0xffff0000 : temp1;
             temp2 = (temp[0] >> 16) & 0x0000FFFF;
             temp2 = (temp2 & 0x00008000) ? temp2 | 0xffff0000 : temp2;
             fft_out[j][i][k].real = (float) temp1;
             fft_out[j][i][k].imag = (float) temp2;
           }  /* for (k...) */
       }  /* for (j...) */
    }  /* for (i...) */
  return;
}
```

A.13 defs.h

```
/* defs.h */

#define NN 64

#define SWAP(a,b) {float swap_temp=(a).real;\
                   (a).real=(b).real;(b).real=swap_temp;\
                   swap_temp=(a).imag;\
                   (a).imag=(b).imag;(b).imag=swap_temp;}

#ifdef IBM
#define log2(x) ((log(x))/(M_LN2))
#define CLK_TICK 100.0
#endif

#ifdef DBLE

#define float double

#endif

typedef struct {
   float real;
   float imag;
```

```
} COMPLEX;

#define AT_LEAST_ARG 2
#define AT_MOST_ARG 4
#define ITERATIONS_ARG 3
#define REPORTS_ARG 4

#define USERTIME(T1,T2)  ((t2.tms_utime-t1.tms_utime)/60.0)
#define SYSTIME(T1,T2)  ((t2.tms_stime-t1.tms_stime)/60.0)
#define USERTIME1(T1,T2)  ((time_end.tms_utime-time_start.tms_utime)/60.0)
#define SYSTIME1(T1,T2)  ((time_end.tms_stime-time_start.tms_stime)/60.0)
#define USERTIME2(T1,T2)  ((end_time.tms_utime-start_time.tms_utime)/60.0)
#define SYSTIME2(T1,T2)  ((end_time.tms_stime-start_time.tms_stime)/60.0)

/* #define LINE_MAX 256 */
#define TRUE 1
#define FALSE 0

/* The following default dimensions are MAXIMUM values. The actual  */
/* dimension for PRIs will be the 1st entry in file "testcase.str". */
/* The 2nd entry in the file is the Target Detection Threshold.     */
/* The rest of the file will contain steering vectors starting with */
/* the main beams then all auxiliary beams. The file "testcase.hdr" */
/* will have descriptions  of entries in "testcase.str" file. Also  */
/* a description of the data file "testcase.dat" will be given. This*/
/* file contains data to fill the input "data_cube[][][]" array.    */

#define COLS 1500  /* Maximum number of columns in holding vector "vec" */
                   /* in house.c for max columns in Householder multiply */
#define MBEAM 12   /* Number of main beams */
#define ABEAM 20   /* Number of auxiliary beams */
#define PWR_BM 9   /* Number of max power beams */
#define V1 64
#define V2 128
#define V3 1500
#define V4 64
#define DIM1 64    /* Number for dimension1 in input data cube */
#define DIM2 128   /* Number for dimension2 in input data cube */
#define DIM3 1500  /* Number for dimension3 in input data cube */
#define DOP_GEN 2048 /* MAX Number of dopplers after FFT */
#define PRI_GEN 2048 /* MAX Number of points for 1500 data points FFT */
                   /*  zero filled after 1500 up to 2048 points */
#define NUM_MAT 32 /* Number of matrices to do householde on */
                   /* NUM_MAT must be less than DIM2 above */

#ifdef APT

#define PRI 1024   /* Number of pris in input data cube */
#define DOP 1024   /* Number of dopplers after FFT */
#define RNG 280    /* Number of range gates in input data cube */
#define EL 32      /* number of elements in input data cube */
#define BEAM 32    /* Number of beams */
#define NUMSEG 7   /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG  /* Number of range gates per segment */
#define RNG_S 320  /* Number sample range gates for 1st step beam forming */
#define DOF EL     /* Number of degrees of freedom */
  extern COMPLEX t_matrix[BEAM][EL];   /* T matrix */

#endif

#ifdef STAP

#define PRI 128   /* Number of pris in input data cube */
#define DOP 128   /* Number of dopplers after FFT */
#define RNG 1024  /* Number of range gates in input data cube */
#define EL 48      /* number of elements in input data cube */
#define BEAM 2     /* Number of beams */
#define NUMSEG 2   /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG /* Number of range gates per segment */
```

```
#define RNG_S 256   /* Number of sample range gates for beam forming */
#define DOF 3*EL    /* Number of degrees of freedom */

#endif

#ifdef GEN
#define EL V4
#define DOF EL

#endif

extern int output_time;    /* Flag if set TRUE, output execution times */
extern int output_report;  /* Flag if set TRUE, output data report files */
extern int repetitions;    /* number of times program has executed */
extern int iterations;     /* number of times to execute program */
```

A.14 compile_hopd

```
mpcc -qarch=pwr2 -O3 -DSTAP -DIBM -o stap bench_mark_STAP.c cell_avg_cfar.c
fft.c fft_STAP.c forback.c cmd_line.c house.c read_input_STAP.c form_beams.c
compute_weights.c compute_beams.c form_str_vecs.c -lm
```

A.15 run.256

```
poe stap /scratch1/masa/new_stap -procs 256 -us
```